GCC Internals: A Conceptual View – Part I

Abhijat Vichare

CFDVS,
Indian Institute of Technology, Bombay

January 2008

Plan

Part I

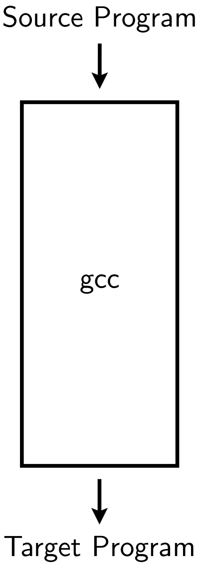
- GCC: Conceptual Structure
- C Program through GCC
- Building GCC

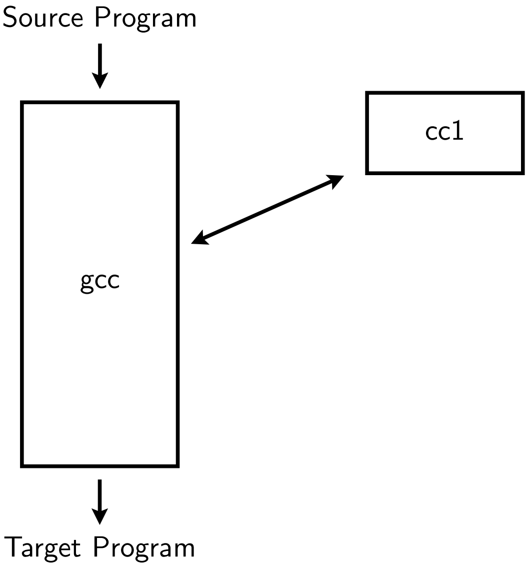
Part II

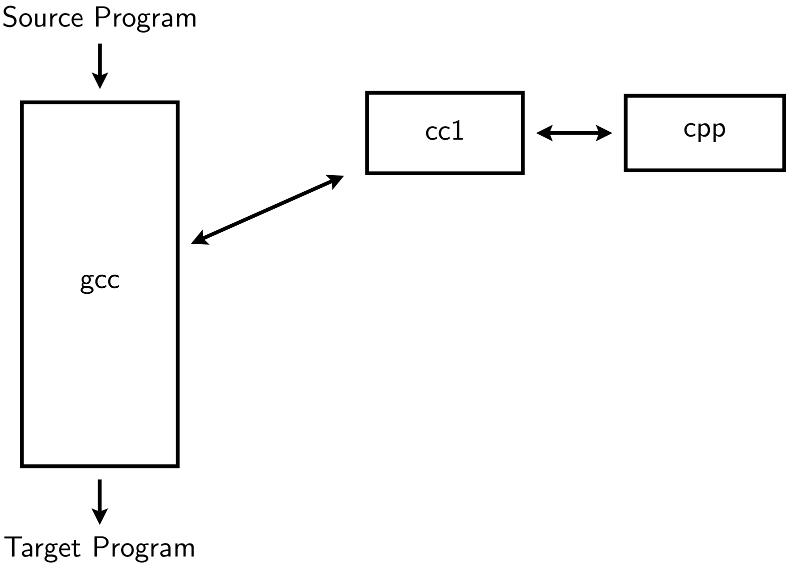
- Gimple
- The MD-RTL and IR-RTL Languages in GCC
- GCC Machine Descriptions

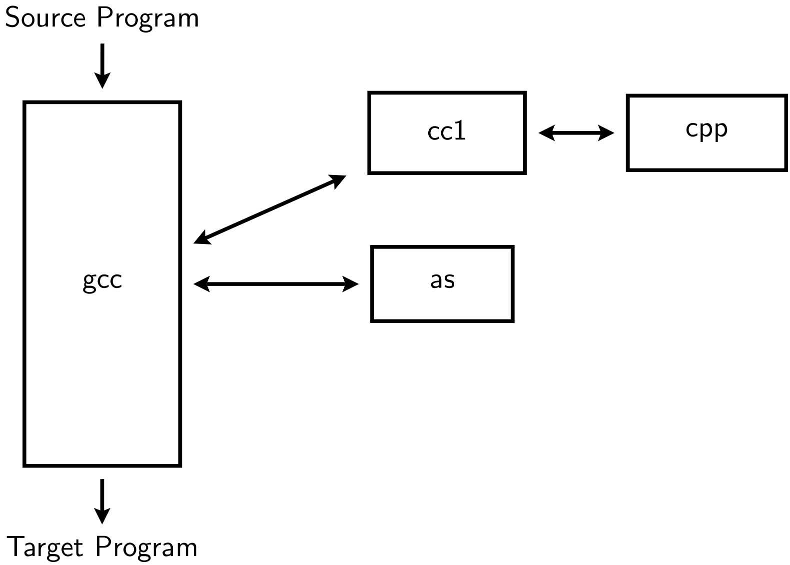
Part I

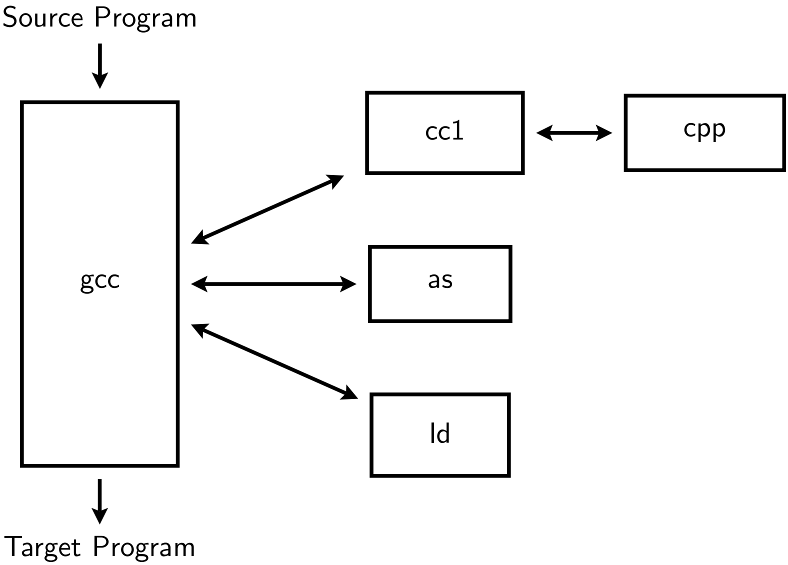
GCC Architecture Concepts

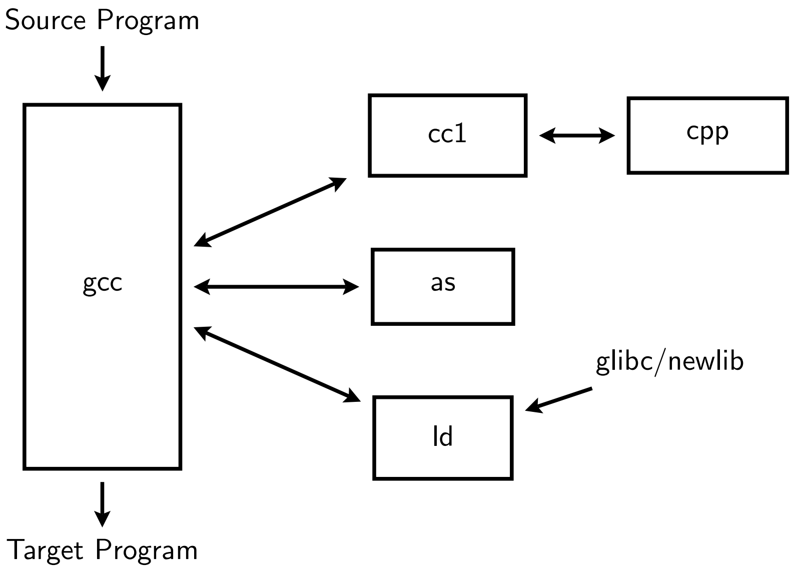

Source Program
gcc
Target Program

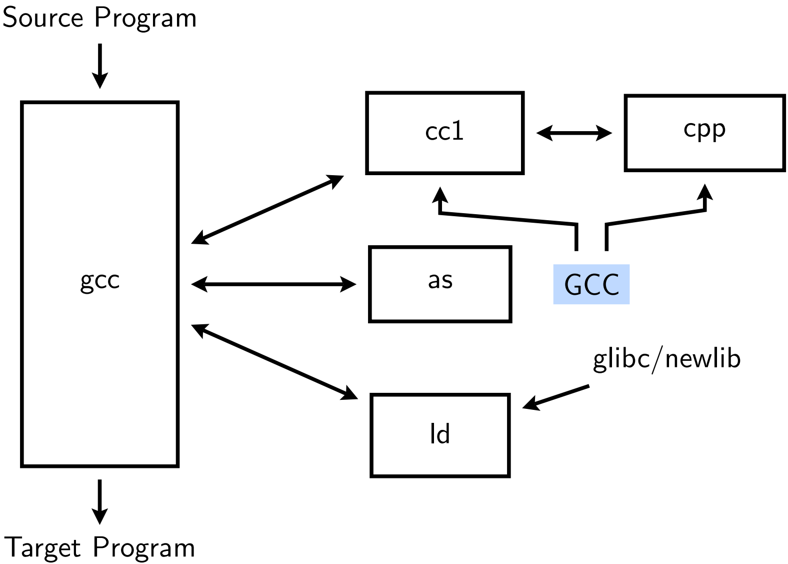

Source Program
gcc
cc1
Target Program


Source Program
gcc
cc1
cpp
Target Program


Source Program
gcc
cc1
cpp
as
Target Program


Source Program
gcc
cc1
cpp
as
ld
Target Program


Source Program
gcc
cc1
cpp
as
glibc/newlib
ld
Target Program


Source Program
gcc
Target Program
cc1
cpp
as
GCC
glibc/newlib
ld

Usual Compilation Phase Sequence vs. GCC

A Typical "Text Book" Compiler Phase Sequence

Parsing	Semantic Analysis	Optimization	Target Code Generation

Usual Compilation Phase Sequence vs. GCC

A Typical "Text Book" Compiler Phase Sequence

Parsing	Semantic Analysis	Optimization	Target Code Generation

GCC is:

- Retargetable: Can generate code for many back ends
- Re-sourcable: Can accept code in many HLLs

Usual Compilation Phase Sequence vs. GCC

A Typical "Text Book" Compiler Phase Sequence

Parsing	Semantic Analysis	Optimization	Target Code Generation

The GCC Phase Sequence looks like

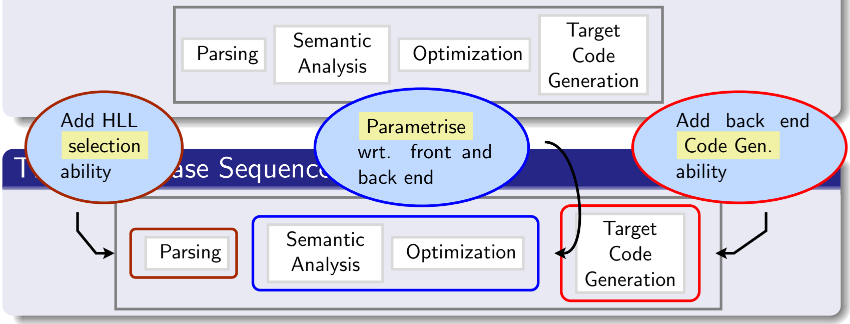
Parsing	Semantic Analysis	Optimization	Target Code Generation

GCC is:

- Retargetable: Can generate code for many back ends
- Re-sourcable: Can accept code in many HLLs

Usual Compilation Phase Sequence vs. GCC

A Typical "Text Book" Compiler Phase Sequence



GCC is:

- Retargetable: Can generate code for many back ends
- Re-sourcable: Can accept code in many HLLs

Implications of Retargetability in GCC

Retargetability

Choose target at $\boxed{\text{build time}}$ than at $\boxed{\text{development time}}$

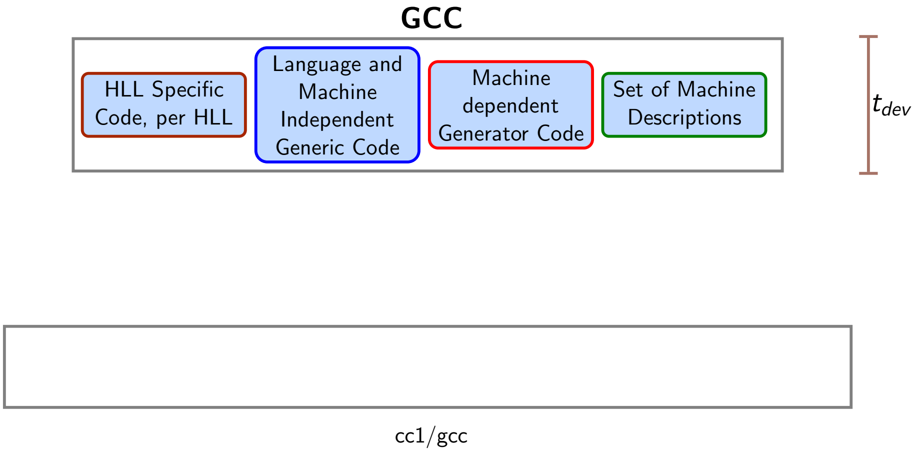
Hence : there are THREE time durations associated with GCC

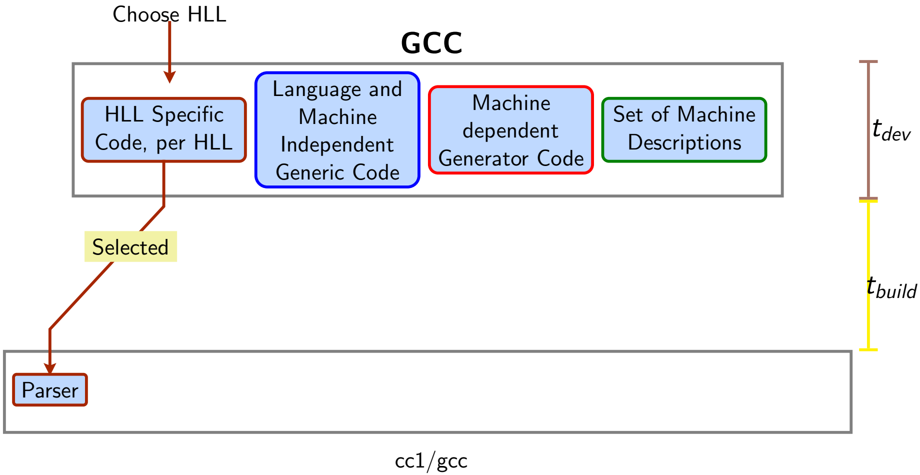
1. $t_{develop}$: The Development time (the "gcc developer" view)
2. t_{build}: The Build time (the "gcc builder" view)
3. t_{op}: The Operation time (the "gcc user" view)

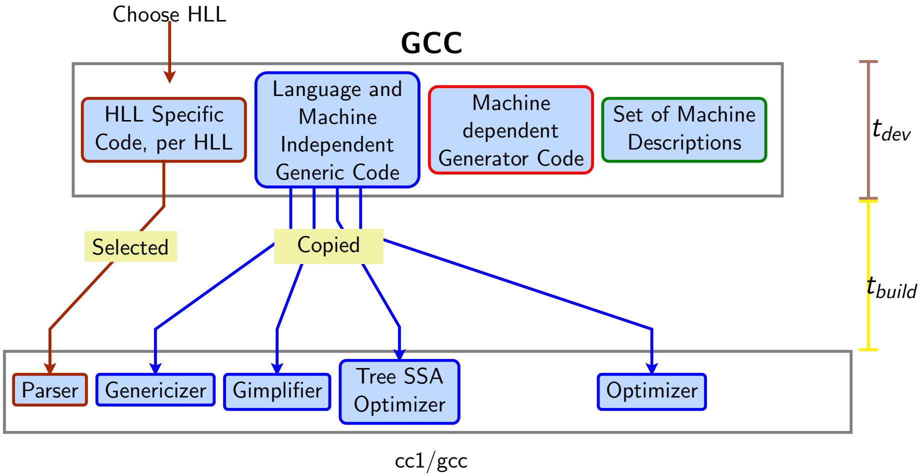
The downloaded GCC sources . . .

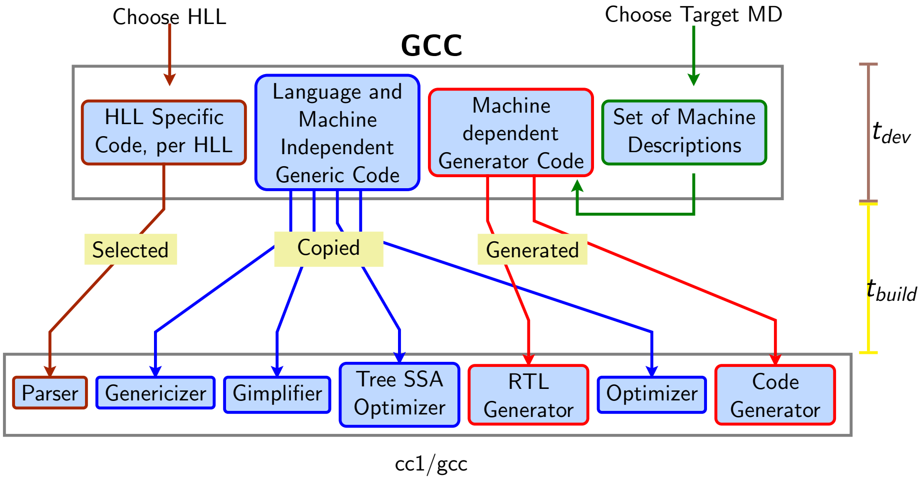
. . . correspond to the "gcc developer" view, and

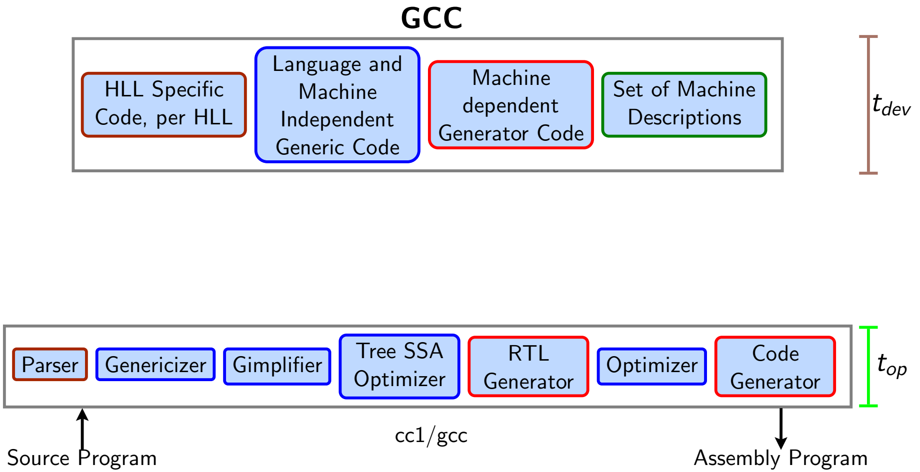
. . . are ready for "gcc builder" view.


GCC
HLL Specific Code, per HLL
Language and Machine Independent Generic Code
Machine dependent Generator Code
Set of Machine Descriptions
t_{dev}
cc1/gcc


Choose HLL
GCC
HLL Specific Code, per HLL
Language and Machine Independent Generic Code
Machine dependent Generator Code
Set of Machine Descriptions
t_{dev}
Selected
t_{build}
Parser
cc1/gcc


Choose HLL
GCC
HLL Specific Code, per HLL
Language and Machine Independent Generic Code
Machine dependent Generator Code
Set of Machine Descriptions
t_{dev}
Selected
Copied
t_{build}
Parser
Genericizer
Gimplifier
Tree SSA Optimizer
Optimizer
cc1/gcc


Choose HLL
Choose Target MD
GCC
HLL Specific Code, per HLL
Language and Machine Independent Generic Code
Machine dependent Generator Code
Set of Machine Descriptions
t_{dev}
Selected
Copied
Generated
t_{build}
Parser
Genericizer
Gimplifier
Tree SSA Optimizer
RTL Generator
Optimizer
Code Generator
cc1/gcc


GCC
HLL Specific Code, per HLL
Language and Machine Independent Generic Code
Machine dependent Generator Code
Set of Machine Descriptions
t_{dev}
Parser
Genericizer
Gimplifier
Tree SSA Optimizer
RTL Generator
Optimizer
Code Generator
t_{op}
Source Program
cc1/gcc
Assembly Program

Is GCC complex?

As a Compiler . . .

- . . . Architecture? – Not quite!
- . . . Implementation? – Very much!

ARCHITECTURE WISE:

1. Superficially: GCC is similar to "typical" compilers!
2. Deeper down: Differences are due to: Retargetability

$\Rightarrow$ GCC can be (and is) used as a Cross Compiler !

IMPLEMENTATION WISE: . . . ? (Next slides)

Some Interesting Facts about GCC 4.0.2

Pristine compiler sources (downloaded tarball)

Lines of C code	1098306
Lines of MD code	217888
Lines of total code	1316194
Total Authors (approx)	63
Backend directories	34

For the targetted (= pristine + generated) C compiler

Total lines of code	810827
Total lines of pure code	606980
Total pure code WITHOUT `#include`	602351
Total number of `#include` directives	4629
Total `#include` files	336

Some Interesting i386 MD Facts

General information

Number of .md files	8
Number of C files	72

Realistic code size information (excludes comments)

Total lines of code	47290
Total lines of .md code	23566
Total lines of header code	9986
Total lines of C code	16961

Part II

C Program through GCC

Conceptually

Input

Practically . . .

The Source

```
int f(char *a)
{
  int n = 10; int i, g;

  i = 0;
  while (i < n) {
    a[i] = g * i + 3;
    i = i + 1;
  }
  return i;
}
```

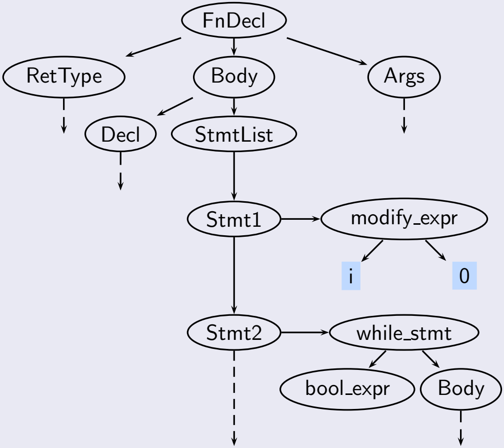
Conceptually

Input

Parse (AST)

Practically ...

Simplified AST



Conceptually

Input

Parse (AST)

IR_1 (Gimple)

Practically ...

Gimple IR

```
f (a)
{
  unsigned int i.0;   char * i.1;
  char * D.1140;      int D.1141;
  ...
  goto <D1136>;
  <D1135>: ...
  D.1140 = a + i.1;
  D.1141 = g * i;
  ...
  <D1136>:
  if (i < n) goto <D1135>;
  ...
}
```

Conceptually

Input

Parse (AST)

IR_1 (Gimple)

Optimization

Practically . . .

Tree SSA form

```
f (a)
{
  ... int D.1144; ...
<bb 0>: n_2 = 10;   i_3 = 0;
  goto <bb 2> (<L1>);
<L0>: ...
  D.1140_9 = a_8 + i.1_7;
  D.1141_11 = g_10 * i_1;
  ...
<L1>:;
  if (i_1 < n_2) goto <L0>;
  else ...;
  ...
}
```

Conceptually

Input

Parse (AST)

IR_1 (Gimple)

Optimization

IR_2 (RTL)

Practically ...

RTL IR (fragment)

```
(insn 21 20 22 2 (parallel [
  (set (reg:SI 61 [ D.1141 ])
    (mult:SI (reg:SI 66)
       (mem/i:SI
         (plus:SI
           (reg/f:SI 54 ...)
           (const_int -8 ...)))))
  (clobber (reg:CC 17 flags))
  ]) -1 (nil)
  (nil))
```

Conceptually

Input

Parse (AST)

IR_1 (Gimple)

Optimization

IR_2 (RTL)

ASM Code

Practically ...

Final ASM (partial)

```
.file "sample.c"
        ...
f:
pushl %ebp
        ...
movl -4(%ebp), %eax
imull -8(%ebp), %eax
addb $3, %al
        ...
leave
ret
        ...
```

```
toplev_main ()                                    toplev.c
 general_init ()                                  toplev.c
 decode_options ()                                toplev.c
 do_compile ()                                    toplev.c
  compile_file()                                  toplev.c
   lang_hooks.parse_file ()                       toplev.c
    c_parse_file ()                             c-parser.c
     c_parser_translation_unit ()               c-parser.c
      c_parser_external_declaration ()          c-parser.c
       c_parser_declaration_or_fndef ()         c-parser.c
        finish_function ()                        c-decl.c
 /* TO: Gimplification */
```

Tip

Use the functions above as breakpoints in gdb on cc1.

Creating GIMPLE representation in cc1 and GCC

```
c_genericize()                                   c-gimplify.c
   gimplify_function_tree()                        gimplify.c
      gimplify_body()                              gimplify.c
         gimplify_stmt()                           gimplify.c
            gimplify_expr()                        gimplify.c
lang_hooks.callgraph.expand_function()
 tree_rest_of_compilation()                   tree-optimize.c
  tree_register_cfg_hooks()                        cfghooks.c
   execute_pass_list()                               passes.c
 /* TO: Gimple Optimisations passes */
```

(Partial) Passes list (tree-optimize.c) (∼ 70 passes)

```
pass_remove_useless_stmts     // Pass
pass_lower_cf                 // Pass
pass_all_optimizations        // Optimiser
   pass_build_ssa             // Optimiser
   pass_dce                   // Optimiser
   pass_loop                  // Optimiser
      pass_complete_unroll    // Optimiser
      pass_loop_done          // Optimiser
   pass_del_ssa               // Optimiser
pass_warn_function_return     // Optimiser
pass_expand                   // RTL Expander
pass_rest_of_compilation      // RTL passes
```

Tree Pass Organisation

- Data structure records pass info: name, function to execute etc. (`struct tree_opt_pass` in `tree-pass.h`)
- Instantiate a `struct tree_opt_pass` variable in each pass file.
- List the pass variables (in `passes.c`).

Dead Code Elimination (`tree-ssa-dce.c`)

```
struct tree_opt_pass pass_dce = {
  "dce",                      // pass name
  tree_ssa_dce,               // fn to execute
  NULL,                       // sub passes
  ...                         // and much more
};
```

- Gimple → non-strict RTL translation
- non-strict RTL passes – information extraction & optimisations
- non-strict → strict RTL passes

```
/* non strict RTL expander pass */
pass_expand_cfg                          cfgexpand.c
 expand_gimple_basic_block ()            cfgexpand.c
  expand_expr_stmt ()                         stmt.c
   expand_expr ()                             stmt.c
/* TO: non strict RTL passes:
 * pass_rest_of_compilation
 */
```

RTL Passes

- Driver: `passes.c:rest_of_compilation ()`
- Basic Structure: Sequence of calls to `rest_of_handle_* ()` + bookkeeping calls. (over 40 calls!)
- Bulk of generated code used here!
  (generated code in: `$GCCBUILDDIR/gcc/*.[ch]`)
- Goals:
  - Optimise RTL
  - Complete the non strict RTL
- Manipulate
  - either the list of RTL representation of input,
  - or contents of an RTL expression,
  - or both.
- Finally: call `rest_of_handle_final ()`

passes.c:rest_of_handle_final() calls

`assemble_start_function ();`	varasm.c
`final_start_function ();`	final.c
`final ();`	final.c
`final_end_function ();`	final.c
`assemble_end_function ();`	varasm.c

Part III

Building GCC

Some Terminology

- The sources of a compiler are compiled (i.e. built) on machine X
  X is called as the Build system
- The built compiler runs on machine Y
  Y is called as the Host system
- The compiler compiles code for target Z
  Z is called as the Target system
- Note: The built compiler itself runs on the Host machine and generates executables that run on Target machine!!!

Some Definitions

Note: The built compiler itself runs on the Host machine and generates executables that run on Target machine!!!

A few interesting permutations of X, Y and Z are:

$X = Y = Z$	Native build
$X = Y \neq Z$	Cross compiler
$X \neq Y \neq Z$	Canadian Cross compiler

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.

Building a Compiler:

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 <u>built</u> 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

⇒ Building cross compilers stops after Stage 1!

GCC Components are:

- Build configuration files
- Compiler sources
- Emulation libraries
- Language Libraries (except C)
- Support software (e.g. garbage collector)

Our conventions

GCC source directory : `$(GCCHOME)`
GCC build directory : `$(GCCBUILDDIR)`
GCC install directory : `$(GCCINSTALLDIR)`

$$\texttt{\$(GCCHOME)} \neq \texttt{\$(GCCBUILDDIR)} \neq \texttt{\$(GCCINSTALLDIR)}$$

Some Information

- Build-Host-Target systems inferred for native builds
- Specify Target system for cross builds
  Build $\equiv$ Host systems: inferred
- Build-Host-Target systems can be explicitly specified too
- For GCC: A "system" $=$ <u>three</u> entities
  - "cpu"
  - "vendor"
  - "os"

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

Basic GCC Building How To

- `prompt$ cd $GCCBUILDDIR`
- `prompt$ configure <options>`
  - Specify target: optional for native builds, necessary for others (option `--target=<host-cpu-vendor string>`)
  - Choose source languages (option `--enable-languages=<CSV lang list (c,java)`)
  - Specify the installation directory (option `--prefix=<absolute path of $(GCCBUILDDIR)>`)

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

Tip

- Run `configure` in `$(GCCBUILDDIR)`.
- See `$(GCCHOME)/INSTALL/`.

Adding a New MD

To add a new backend to GCC

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

  for recognized system names.

Tip

Read comments in `$GCCHOME/config.sub` & `$GCCHOME/gcc/config/<cpu>`.

GCC builds in two main phases:

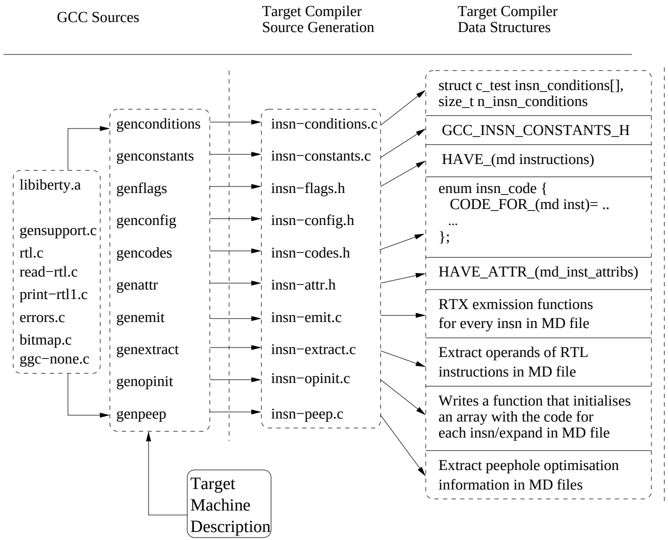
- Adapt the compiler source for the specified build/host/target systems
  Consider a cross compiler:
  - Find the target MD in the source tree
  - "Include" MD info into the sources (details follow)
- Compile the adapted sources
- NOTE:
  - Incomplete MD specifications $\Rightarrow$ Unsuccessful build
  - Incorrect MD specification $\Rightarrow$ Run time failures/crashes (either ICE or `SIGSEGV`)

- make first compiles and runs a series of programs that process the target MD
- Typically, the program source file names are prefixed with gen
- The $GCCHOME/gcc/gen*.c programs
  - read the target MD files, and
  - extract info to create & populate the main GCC data structures

Example

Consider genconstants.c:

- <target>.md may define UNSPEC_* constants.
- genconstants.c – reads UNSPEC_* constants
- genconstants.c – generates corresponding #defines
- Collect then into the insn-constants.h
- #include "insn-constants.h" in the main GCC sources


GCC Sources
Target Compiler Source Generation
Target Compiler Data Structures
libiberty.a
gensupport.c
rtl.c
read−rtl.c
print−rtl1.c
errors.c
bitmap.c
ggc−none.c
genconditions
genconstants
genflags
genconfig
gencodes
genattr
genemit
genextract
genopinit
genpeep
insn−conditions.c
insn−constants.c
insn−flags.h
insn−config.h
insn−codes.h
insn−attr.h
insn−emit.c
insn−extract.c
insn−opinit.c
insn−peep.c
struct c_test insn_conditions[],
size_t n_insn_conditions
GCC_INSN_CONSTANTS_H
HAVE_(md instructions)
enum insn_code {
CODE_FOR_(md inst)= ..
...
};
HAVE_ATTR_(md_inst_attribs)
RTX exmission functions
for every insn in MD file
Extract operands of RTL
instructions in MD file
Writes a function that initialises
an array with the code for
each insn/expand in MD file
Extract peephole optimisation
information in MD files
Target
Machine
Description

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
```