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
The GNU Tool Chain

Source Program

\[ \text{gcc} \]

Target Program
The GNU Tool Chain

Source Program

gcc

Target Program

cc1
The GNU Tool Chain

Source Program

gcc

c1

cpp

Target Program
The GNU Tool Chain

Source Program

gcc

Target Program

cc1

cpp

as

ld
The GNU Tool Chain

Source Program

gcc

Target Program

cc1

cpp

as

ld

glibc/newlib

GCC
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

- Parsing
- Semantic Analysis
- Optimization
- Target Code Generation

The GCC Phase Sequence looks like

- Parsing
- Semantic Analysis
- Optimization
- Target Code Generation

Add HLL selection ability

Parametrise wrt. front and back end

Add back end Code Gen. ability

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 **build time** than at **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.
The GCC Compiler Generation Framework

GCC

- HLL Specific Code, per HLL
- Language and Machine Independent Generic Code
- Machine dependent Generator Code
- Set of Machine Descriptions

$t_{dev}$

cc1/gcc
The GCC Compiler Generation Framework

Choose HLL

GCC

HLL Specific Code, per HLL
Language and Machine Independent Generic Code
Machine dependent Generator Code
Set of Machine Descriptions

Selected

Parser

cc1/gcc
The GCC Compiler Generation Framework

Choose HLL

HLL Specific Code, per HLL

Language and Machine Independent Generic Code

Machine dependent Generator Code

Set of Machine Descriptions

Selected

Copied

Parser

Genericizer

Gimplifier

Tree SSA Optimizer

Optimizer

cc1/gcc

t_{dev}

t_{build}
The GCC Compiler Generation Framework

GCC

Choose HLL

HLL Specific Code, per HLL

Selected

Language and Machine Independent Generic Code

Copied

Machine dependent Generator Code

Generated

Set of Machine Descriptions

Choose Target MD

Parser

Genericizer

Gimplitifier

Tree SSA Optimizer

RTL Generator

Optimizer

Code Generator

cc1/gcc

A.Vichare

GCC Internals
The GCC Compiler Generation Framework

GCC

- HLL Specific Code, per HLL
- Language and Machine Independent Generic Code
- Machine dependent Generator Code
- Set of Machine Descriptions

Parser  Genericizer  Gimplifier  Tree SSA Optimizer  RTL Generator  Optimizer  Code Generator

Source Program  cc1/gcc  Assembly Program

\[ t_{dev} \]

\[ t_{op} \]
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

⇒ 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)

<table>
<thead>
<tr>
<th>Metric</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Lines of C code</td>
<td>1,098,306</td>
</tr>
<tr>
<td>Lines of MD code</td>
<td>217,888</td>
</tr>
<tr>
<td>Lines of total code</td>
<td>1,316,194</td>
</tr>
<tr>
<td>Total Authors (approx)</td>
<td>63</td>
</tr>
<tr>
<td>Backend directories</td>
<td>34</td>
</tr>
</tbody>
</table>

For the targeted (= pristine + generated) C compiler

<table>
<thead>
<tr>
<th>Metric</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Total lines of code</td>
<td>810,827</td>
</tr>
<tr>
<td>Total lines of pure code</td>
<td>606,980</td>
</tr>
<tr>
<td>Total pure code WITHOUT <code>#include</code></td>
<td>602,351</td>
</tr>
<tr>
<td>Total number of <code>#include</code> directives</td>
<td>4,629</td>
</tr>
<tr>
<td>Total <code>#include</code> files</td>
<td>336</td>
</tr>
</tbody>
</table>
### General information

<table>
<thead>
<tr>
<th>Description</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Number of <code>.md</code> files</td>
<td>8</td>
</tr>
<tr>
<td>Number of C files</td>
<td>72</td>
</tr>
</tbody>
</table>

### Realistic code size information (excludes comments)

<table>
<thead>
<tr>
<th>Description</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Total lines of code</td>
<td>47290</td>
</tr>
<tr>
<td>Total lines of <code>.md</code> code</td>
<td>23566</td>
</tr>
<tr>
<td>Total lines of header code</td>
<td>9986</td>
</tr>
<tr>
<td>Total lines of C code</td>
<td>16961</td>
</tr>
</tbody>
</table>
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;
}
C Program: Journey through GCC

**Conceptually**
- Input
- Parse (AST)

**Practically...**

**Simplified AST**

```
FnDecl
  └── RetType
  └── Body
    └── Args
        ├── Decl
        └── StmtList
            ├── Stmt1
            │   └── modify_expr
            │       └── i
            │           └── 0
            └── Stmt2
                └── while_stmt
                    ├── bool_expr
                    └── Body
```
C Program: Journey through GCC

**Conceptually**
- Input
- Parse (AST)
- IR₁ (Gimple)

**Practically ...**

**Gimple IR**

```c
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>;
    ...
}
```
C Program: Journey through GCC (1:1:6,7)

Conceptually

Input
Parse (AST)
IR₁ (Gimple)
Optimization

Practically ...

Tree SSA form

```c
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 ...;
    ...
}
```

A.Vichare  GCC Internals
C Program: Journey through GCC

Conceptually

- Input
- Parse (AST)
- IR₁ (Gimple)
- Optimization
- IR₂ (RTL)

Practically ...

**RTL IR (fragment)**

Conceptually

Input
Parse (AST)
IR₁ (Gimple)
Optimization
IR₂ (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
...
Front End Processing Sequence in cc1 and GCC

```
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

<table>
<thead>
<tr>
<th>Function</th>
<th>Source File</th>
</tr>
</thead>
<tbody>
<tr>
<td>c_genericize()</td>
<td>c-gimplify.c</td>
</tr>
<tr>
<td>gimplify_function_tree()</td>
<td>gimplify.c</td>
</tr>
<tr>
<td>gimplify_body()</td>
<td>gimplify.c</td>
</tr>
<tr>
<td>gimplify_stmt()</td>
<td>gimplify.c</td>
</tr>
<tr>
<td>gimplify_expr()</td>
<td>gimplify.c</td>
</tr>
<tr>
<td>lang_hooks.callgraph.expand_function()</td>
<td></td>
</tr>
<tr>
<td>tree_rest_of_compilation()</td>
<td>tree-optimize.c</td>
</tr>
<tr>
<td>tree_register_cfg_hooks()</td>
<td>cfghooks.c</td>
</tr>
<tr>
<td>execute_pass_list()</td>
<td>passes.c</td>
</tr>
</tbody>
</table>

/* TO: Gimple Optimisations passes */
The Tree passes list

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

<table>
<thead>
<tr>
<th>Pass Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>pass_remove_useless_stmts</td>
<td>Pass</td>
</tr>
<tr>
<td>pass_lower_cf</td>
<td>Pass</td>
</tr>
<tr>
<td>pass_all_optimizations</td>
<td>Optimiser</td>
</tr>
<tr>
<td>pass_build_ssa</td>
<td>Optimiser</td>
</tr>
<tr>
<td>pass_dce</td>
<td>Optimiser</td>
</tr>
<tr>
<td>pass_loop</td>
<td>Optimiser</td>
</tr>
<tr>
<td>pass_complete_unroll</td>
<td>Optimiser</td>
</tr>
<tr>
<td>pass_loop_done</td>
<td>Optimiser</td>
</tr>
<tr>
<td>pass_del_ssa</td>
<td>Optimiser</td>
</tr>
<tr>
<td>pass_warn_function_return</td>
<td>Optimiser</td>
</tr>
<tr>
<td>pass_expand</td>
<td>RTL Expander</td>
</tr>
<tr>
<td>pass_rest_of_compilation</td>
<td>RTL passes</td>
</tr>
</tbody>
</table>
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)

```c
struct tree_opt_pass pass_dce = {
    "dce", // pass name
    tree_ssa_dce, // fn to execute
    NULL, // sub passes
    ... // and much more
};
```
Gimple $\rightarrow$ non-strict RTL translation
non-strict RTL passes – information extraction & optimisations
non-strict $\rightarrow$ strict RTL passes

/* non strict RTL expander pass */
pass_expand_cfg
expand_gimple_basic_block ()
expand_expr_stmt ()
expand_expr ()

/* TO: non strict RTL passes: */
* pass_rest_of_compilation
*/
• 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 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

⇒ 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)

$(GCCHOME) ≠ $(GCCBUILDDIR) ≠ $(GCCINSTALLDIR)
Some Information

- 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
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/.
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 ⇒ Unsuccessful build
- Incorrect MD specification ⇒ 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
### The GCC Build Process
**Adapting the Compiler Sources – Pictorial view**

#### GCC Sources
- libiberty.a
- gensupport.c
- rtl.c
- read-rtl.c
- print-rtl1.c
- errors.c
- bitmap.c
- gcc-none.c

#### Target Machine Description
- genconditions
- genconstants
- genflags
- genconfig
- gencodes
- genattr
- genemit
- genextract
- genopinit
- genpeep

#### Target Compiler Source Generation
- 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

#### Target Compiler Data Structures
- 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 transmission 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 file
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