

# GCC Internals: A Conceptual View

Abhijat Vichare

CFDVS,  
Indian Institute of Technology, Bombay

FOSS.IN  
December 2007



# Part I

## Goals

# Documentation is as important as coding, if not more!

## The Central Message

Programming = document + code

Please contribute to documenting code too!!

## How am I going to talk you into it :)

- Share our experiences
- Share with you the GCC Model that we have developed
- Talk about what next!
  - The code base
  - The docs base

## About GCC

- **Standard** system compiler for GNU/Linux
- **Optional** compiler for many other systems
- **Highly retargetable**, and well implemented code
- Industry strength, and **standards compliant**
- **Critical** component of almost all software, esp. system software.

## About it's Internals

- Not many know the *architecture*
- Official internals docs describe many details

## Why Document the Internals?

- the TODAY
  - Programmer “self reference”: *Have I really expressed in code what I want to do?*
  - GCC Community “reference”: *Are the ideas meshing well?*
  - FOSS Community: *What parts of GCC can impact project X, it at all?*
  - And ... a general knowledge base!
- the TOMORROW
  - Serve as a “reference implementation” platform for new research
  - **Help**: from newbie → GCC hacker
  - Better **evolution** of GCC in future

Which means ...

Efficient use of the collective time

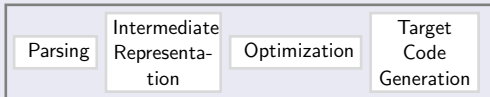
# So? What are we going to talk about GCC Internals?

But first: What are we not going to talk about

- Front end specific part; but C – an illustrative HLL
- Standards compliance
- Allied architecture issues: error handling, memory management etc.
- Parsing and Optimizations

We will talk about

Usual Compilation Phase Seq → GCC's



A Typical "Text Book" Compiler Phase Sequence

GCC

- Architecture
- Retargetability
- IRs
- Contribute

## Part II

# GCC: The GNU Compiler Collection

# The THREE Time Durations

## Retargetability

Choose target at `build time` than at `development time`

**Hence** : there are THREE time durations associated with GCC

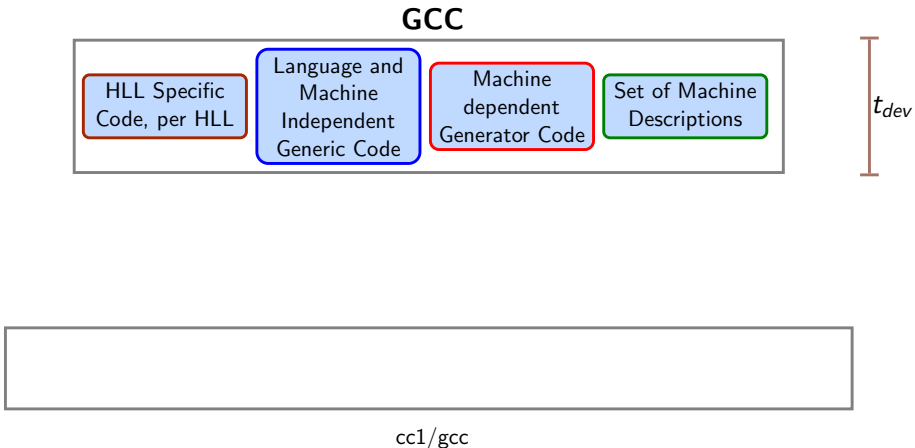
- 1  $t_{dev}$ : The **Development** time (the “gcc developer” view)
- 2  $t_{build}$ : The **Build** time (the “gcc builder” view)
- 3  $t_{op}$ : The **Operation** time (the “gcc user” view)

## Additionally

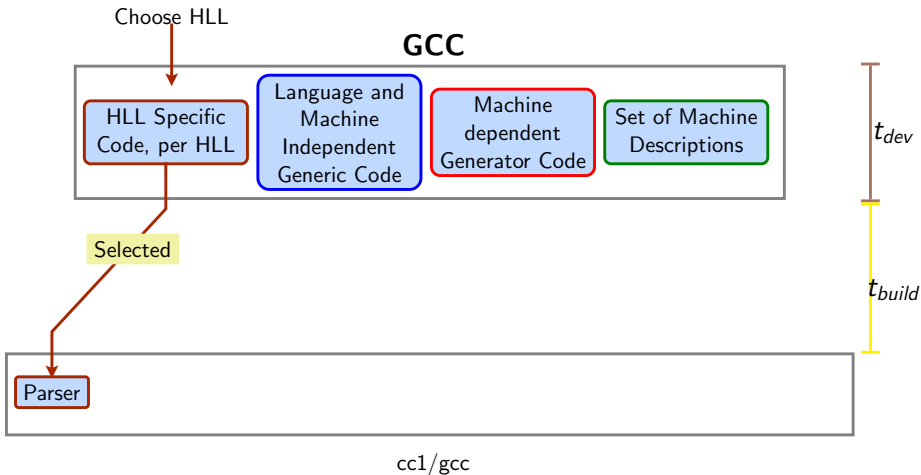
We may split  $t_{build}$  into

- $t_{gen}$  – the “generation” time
- $t_{comp}$  – the “compilation” time.

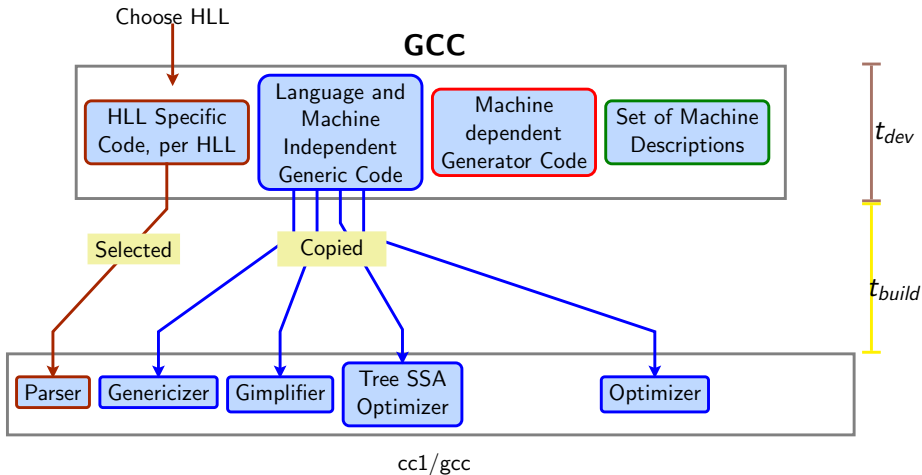
# The GCC Compiler Generation Framework



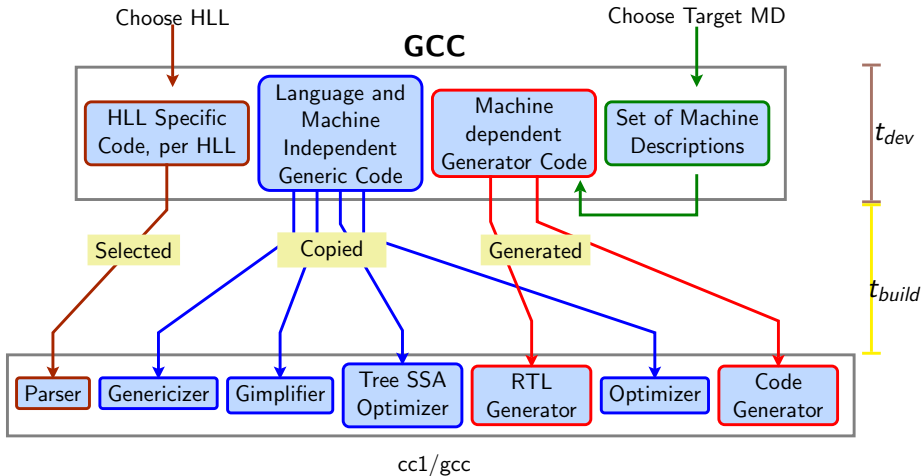
# The GCC Compiler Generation Framework



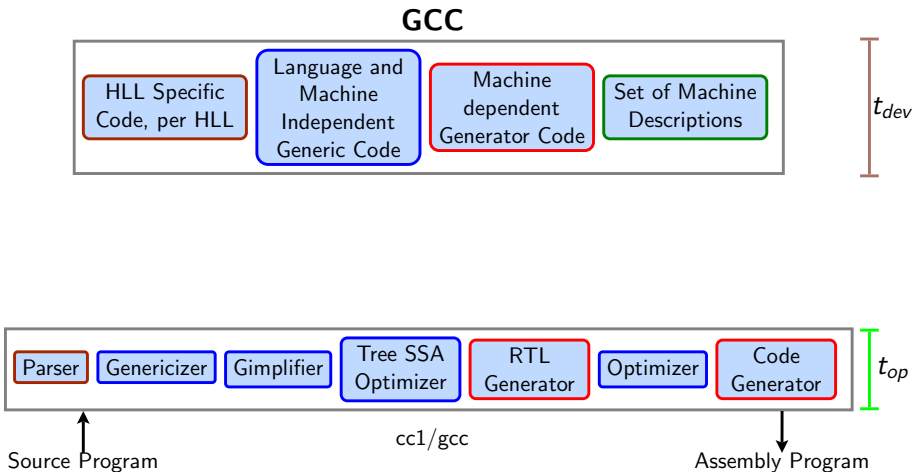
# The GCC Compiler Generation Framework



# The GCC Compiler Generation Framework



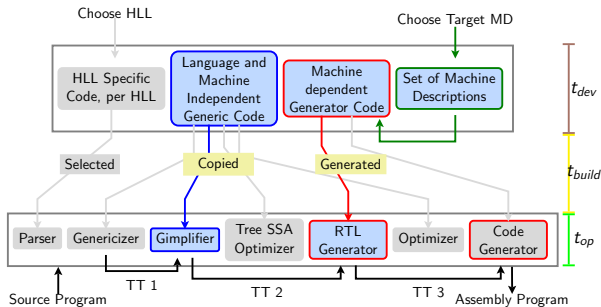
# The GCC Compiler Generation Framework



# The GCC Translation Tables

To compile from one GCC IR to another ...

- Define a translation table (TT) (aka finite function (FF))
- Each object of the “source” IR is mapped to expression in terms of “target” IR objects
- The mapping preserves semantics

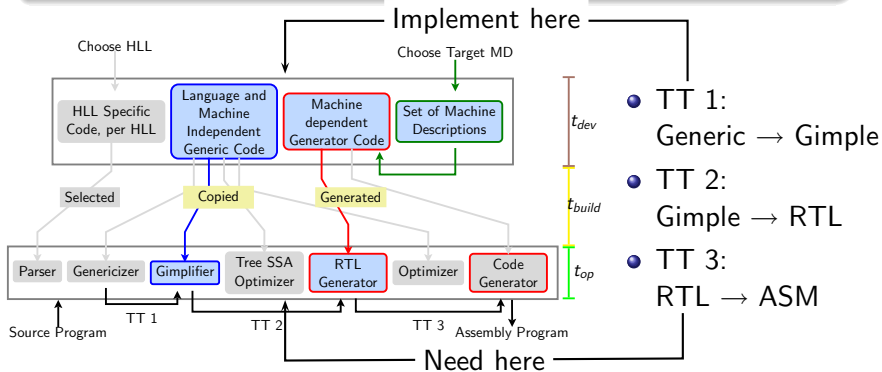


- TT 1: Generic → Gimple
- TT 2: Gimple → RTL
- TT 3: RTL → ASM

# The GCC Translation Tables

To compile from one GCC IR to another ...

- Define a translation table (TT) (aka finite function (FF))
- Each object of the “source” IR is mapped to expression in terms of “target” IR objects
- The mapping preserves semantics



# Retargetability as GCC Does It

Retargetability: Choose target at  $t_{build}$  than at  $t_{dev}$

- ( $t_{dev}$ ) Identify compiler parts influenced by target
- ( $t_{dev}$ ) “Parametrize” these parts
- ( $t_{dev}$ ) Separate their definition and use
- ( $t_{build}$ ) Choose particular target
- ( $t_{build}$ ) Generate target specific compiler parts
- ( $t_{build}$ ) Join the separated parts

Implement “parametrization” by

- Using C Preprocessor macros to name values, and
- Using a suitable language (i.e. RTL) to capture target instruction semantics

## Part III

# The GCC Gimple IR

## The Goals of GIMPLE are

- Lower control flow  
Program = sequenced statements + unrestricted jump
- Simplify expressions, introduce temporary variables as needed  
Typically: two operand assignments!
- Simplify scope  
move local scope to block begin – including temporaries

## Notice

Lowered control flow → nearer to register machines + Easier SSA!

## Tree manipulation passes (`tree-optimize.c`)

- **Gimplifier** case analyzes GENERIC nodes, calls corresponding gimplifier.  
 $\{\text{Gimple}\} = \{\text{AST/Generic}\} - \{\text{Control flow nodes}\}$
- **Node type** specific gimplifiers
- **Optimization passes** on tree representation, and
- **Translate** to next IR, i.e. RTL
  - Depth first traverse the “input” Gimple representation
  - Generate a linear list RTL representation

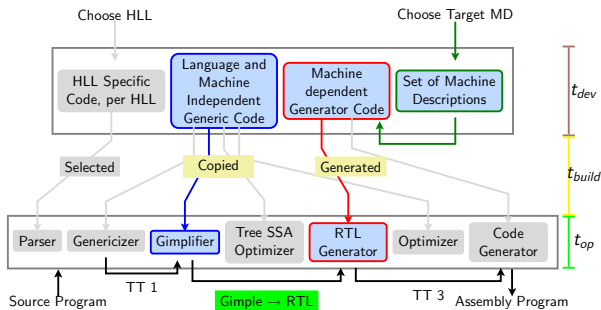
BUT WE HAVE A PROBLEM ...

# Gimple → RTL Translation Table – Part I

**PROBLEM:** Gimple (m/c indep.) → RTL (m/c specific)!

**TO DO :** Implement m/c indep. to m/c dep. translation at  $t_{dev}$

**GIVEN :** the actual target will be known only at  $t_{build}$

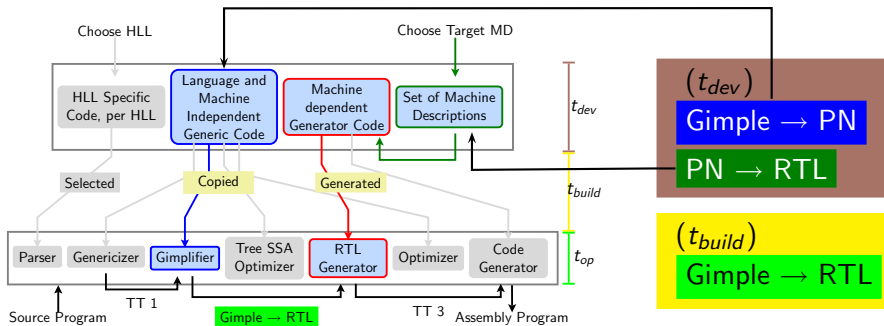


# Gimple → RTL Translation Table – Part I

**PROBLEM:** Gimple (m/c indep.) → RTL (m/c specific)!

**TO DO:** Implement m/c indep. to m/c dep. translation at  $t_{dev}$

**GIVEN:** the actual target will be known only at  $t_{build}$



# Target indep. rep. to target dep. rep. in GCC

MODIFY\_EXPR

(set (<dest>) (<src>))

# Target indep. rep. to target dep. rep. in GCC



# Target indep. rep. to target dep. rep. in GCC

MODIFY\_EXPR

"movsi"

(set (<dest>) (<src>))

Standard Pattern Name

Separate to generic code and MD

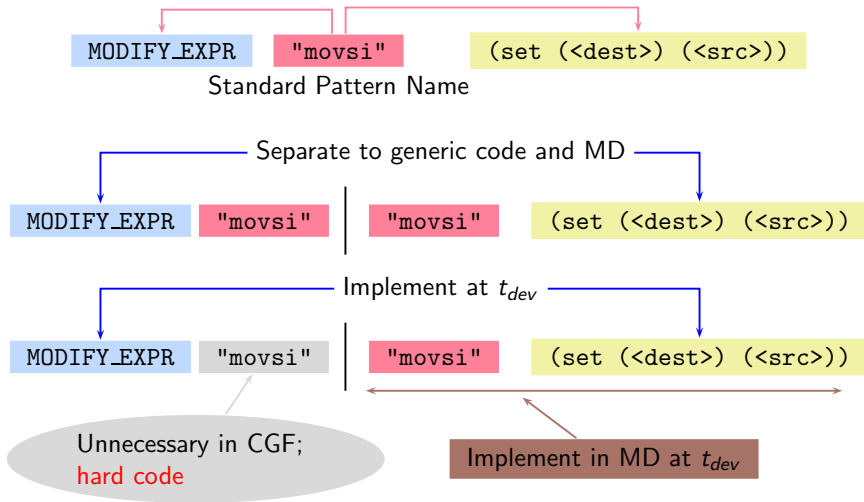
MODIFY\_EXPR

"movsi"

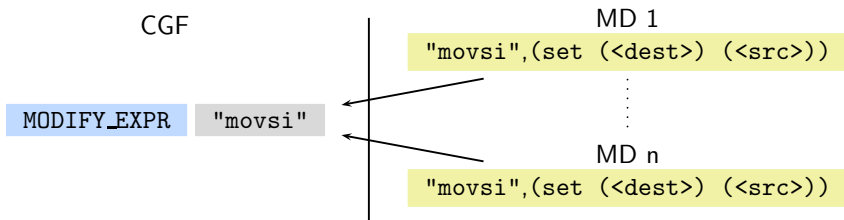
"movsi"

(set (<dest>) (<src>))

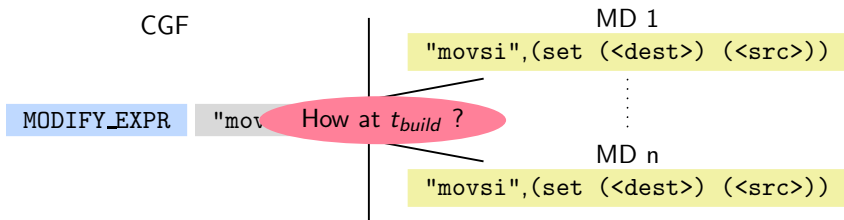
# Target indep. rep. to target dep. rep. in GCC



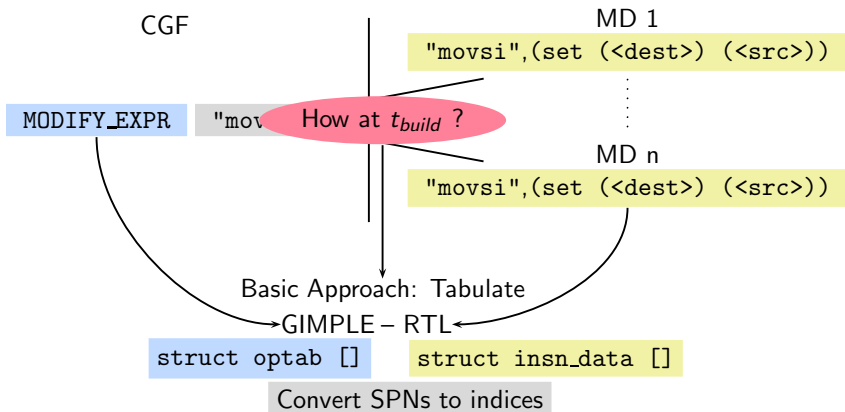
# Retargetability $\Rightarrow$ Multiple MD vs. One CGF!



# Retargetability $\Rightarrow$ Multiple MD vs. One CGF!



# Retargetability $\Rightarrow$ Multiple MD vs. One CGF!



## Part IV

# The GCC RTL IR

## An Introduction

- **External syntax** & **internal structure** inspired by LISP lists
  - **Internal**: rtx structs pointing to other structs
  - **External**: Lisp like text – lots of parentheses
- Two uses:
  - In MD: Specify target instruction semantics at  $t_{dev}$
  - As IR: Represent input program at  $t_{op}$
- We have identified three kinds:
  - MD constructs : Used in MD (e.g. `define_insn`)
  - Operators : Used in MD and IR (e.g. `set`, `plus`)
  - IR (or FR) constructs : Used in IR (e.g. `insn`, `jump_insn`)

See GCC online docs at

<http://gcc.gnu.org/onlinedocs/gccint/RTL.html>

# RTL Goals and Use

Goal 1: Specify target instruction semantics at  $t_{dev}$

Capture target instruction semantics as RTXs

Use MD constructs and operators

Goal 2: Represent input program at  $t_{op}$

- Lower data
- “Express” the “captured” target semantics in IR
- Goal: Every RTX of last RTL pass = unique ASM string.

Use IR constructs and operators

## Notice

Lowered data and procedures → nearer to typical hardware

## At development time $t_{dev}$

- list RTL objects in `$GCCHOME/gcc/rtl.def`
- define the data structure for RTL objects
- use MD RTL constructs and RTL operators to express target instruction semantics in the GCC MD
- Implement code to process the RTL IR at  $t_{op}$

## Alert: At $t_{build}$

Convert (text) RTL at  $t_{dev}$  to RTL data structures and compile.

```
(define_insn
  "movsi"
  (set
    (match_operand 0 "register_operand" "r")
    (match_operand 1 "const_int_operand" "k")
  )
  "" /* C boolean expression, if required */
  "mov %0, %1"
)
```

# RTL at $t_{dev}$ : Example of Specifying target inst. semantics

Define new inst. pattern

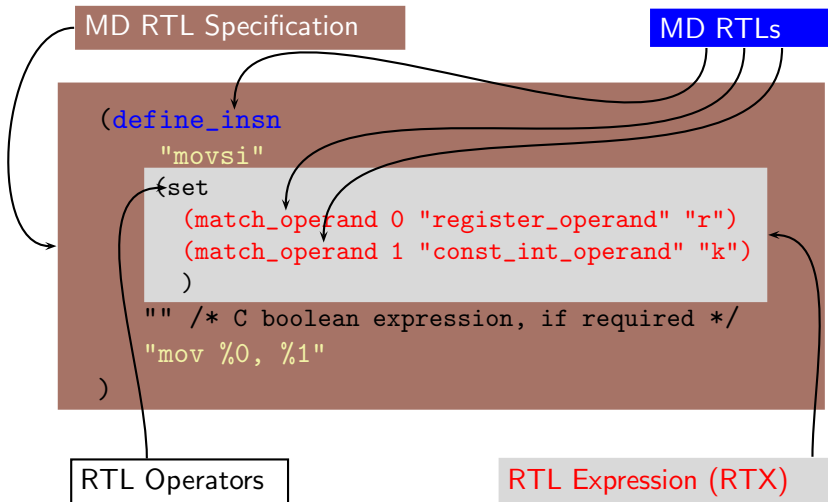
(Standard) Pattern Name

```
(define_insn  
  "movsi"  
  (set  
    (match_operand 0 "register_operand" "r")  
    (match_operand 1 "const_int_operand" "k")  
  )  
  "" /* C boolean expression, if required */  
  "mov %0, %1")
```

RTL: Capture semantics  
of target inst.

target asm inst. =  
Concrete syntax for RTL

# RTL at $t_{dev}$ : Example of Specifying target inst. semantics



# RTL at $t_{build}$ : Gimple $\rightarrow$ RTL Translation Table – Part II

Conversion of RTL from Textual to Internal Form

```
(define_insn
```

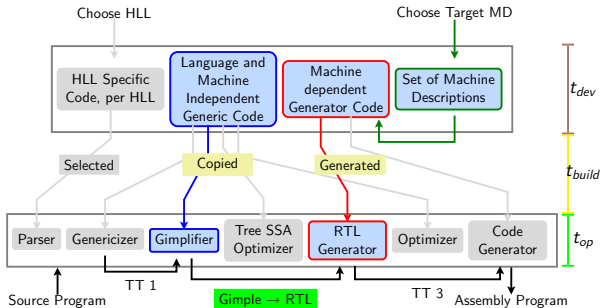
```
  "movsi"
```

```
  (set (op0) (op1))
```

```
  ""
```

```
  "mov %0, %1")
```

$t_{dev}$



# RTL at $t_{build}$ : Gimple $\rightarrow$ RTL Translation Table – Part II

Conversion of RTL from Textual to Internal Form

```
(define_insn
  "movsi"
  (set (op0) (op1))
  ""
  "mov %0, %1")
```

$t_{dev}$

```
rtl
gen_movsi (rtl operand0, rtl operand1)
{
  ...
  emit_insn (gen_rtx_SET (VOIDmode, op0, op1));
  ...
}
```

$t_{build}$

# RTL at $t_{build}$ : Gimple $\rightarrow$ RTL Translation Table – Part II

Conversion of RTL from Textual to Internal Form

```
(define_insn  
  "movsi"  
  (set (op0) (op1))  
  ""  
  "mov %0, %1")
```

$t_{dev}$

```
rtl  
gen_movsi (rtl operand0, rtl operand1)  
{  
  ...  
  emit_insn (gen_rtx_SET (VOIDmode, op0, op1));  
  ...  
}
```

$t_{build}$

# RTL at $t_{build}$ : Gimple $\rightarrow$ RTL Translation Table – Part II

Conversion of RTL from Textual to Internal Form

```
(define_insn  
  "movsi"  
  (set (op0) (op1))  
  ""  
  "mov %0, %1")
```

$t_{dev}$

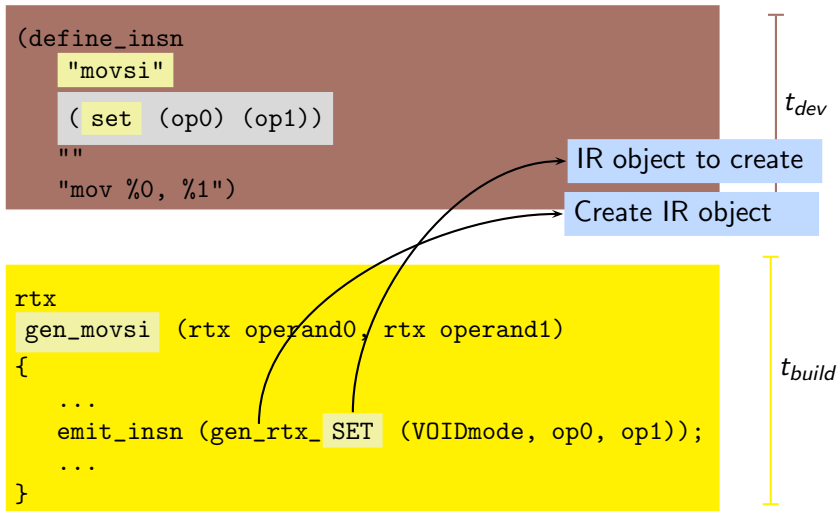
IR object to create

```
rtx  
gen_movsi (rtx operand0, rtx operand1)  
{  
  ...  
  emit_insn (gen_rtx_SET (VOIDmode, op0, op1));  
  ...  
}
```

$t_{build}$

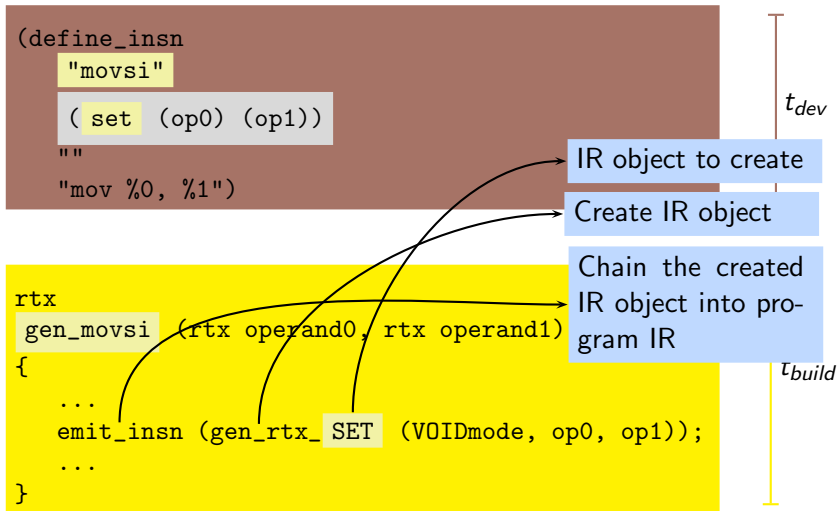
# RTL at $t_{build}$ : Gimple $\rightarrow$ RTL Translation Table – Part II

Conversion of RTL from Textual to Internal Form



# RTL at $t_{build}$ : Gimple $\rightarrow$ RTL Translation Table – Part II

Conversion of RTL from Textual to Internal Form



# RTL at $t_{op}$ : Example of RTL in use as IR

```
(insn 24 22 25 1
```

```
(set  
  (reg:SI 58 [D.1283])  
  (const_int 0 [0x0])  
)
```

```
-1
```

```
(nil)
```

```
(nil)
```

```
)
```

IR RTL 1

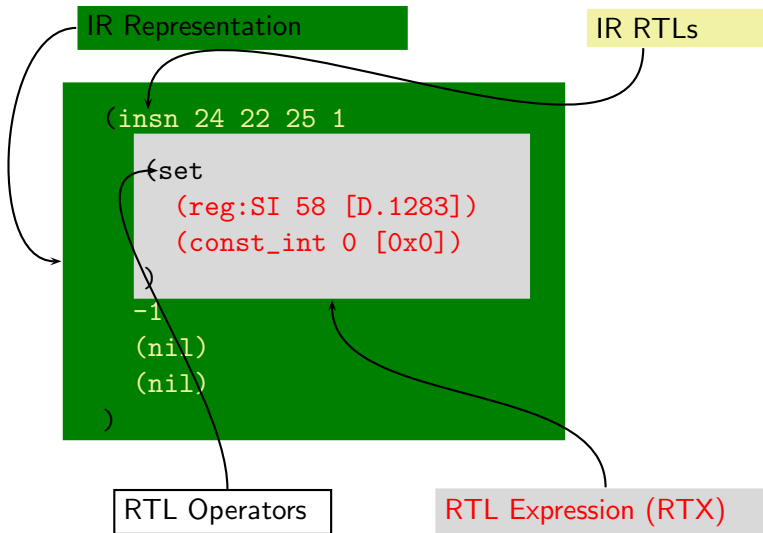
IR RTL ...

IR RTL 24

IR RTL ...

IR RTL n

# RTL at $t_{op}$ : Example of RTL in use as IR



# RTL at $t_{op}$ : Example of RTL in use as IR

Instruction position in sequence

Previous inst.

Next inst.

```
(insn 24 22 25 1
```

```
(set  
  (reg:SI 58 [D.1283])  
  (const_int 0 [0x0])  
)
```

```
-1
```

```
(nil)
```

```
(nil)
```

```
)
```

Instantiation of RTX specified in MD.

Register #58 matches the corresponding match\_operand spec.

Similarly, for the next operand.

# RTL at $t_{op}$ : Example of RTL in use as IR

```
(insn 24 22 25 1
  (set
    (reg:SI 58 [D.1283])
    (const_int 0 [0x0])
  )
  -1
  (nil)
  (nil)
)
```

Note:

RTL is incomplete.

Hard register for  
pseudoregister #58  
unknown (yet).

Allocate: #58 = eax

RTL is now complete.

Instantiated RTL is target dependent because the RTL specified in MD captures target instruction semantics.

# How has this description been useful?

For MD RTL and RTL operators, i.e. for writing MD, we have

- LALR(1) grammar
- `<target>.md` Syntax checker available
- Syntax based editing support

Emacs “GCC MD” mode as a proof-of-concept available

To Do:

- Grammar based “completion” and “help”
- Template Instantiation: non strict RTX of a template

## Some R&D / hacking ideas

- RTL interpreters ?
- IRs as Abstract Machines ?
- Formal verification ?

Part V

To Contribute

## The Central Message

Documentation is a part of software development

## GCC Internals Documentation: Current state

- Programmer's reference: comments in code – EXISTS
- Most of GCC API (e.g. support libraries API) – EXISTS
- GCC Internals Architecture – PARTIALLY DONE!

## Documentation Goal: Explain how things fit together

- We have addressed: How basic GCC works & how to port it!
  - Logically organized source file groups
  - Basic phase sequence and lowering operations
  - Basic tree and RTL data structures

# The To Do's – Part I

## SOME DOCUMENTATION TO DO's – mainly the details

- Front end architecture (we have ignored them :( )
- Detailing, e.g. Back end RTL data structures
- Advanced control flow architecture: e.g. exception handling
- Describe architecture of support libraries
  - Emulation libraries: `libgcc`
  - HLL runtime libraries: `libjava`, `libfortran` etc.
  - External libraries: interface and architecture (e.g. `glibc`)
- Testing: the infrastructure, features tested, test results
- Many critical internal details – e.g. the reload pass.

## See

<http://gcc.gnu.org/projects/documentation.html>

<http://gcc.gnu.org/wiki>

# The To Do's – Part II

## SOME CODE TO DO'S

- Many active development branches
- Improve machine description
- Better instruction selection
- Inter procedure analysis
- Support for multi core processors, DSPs etc.
- Additional front ends

See

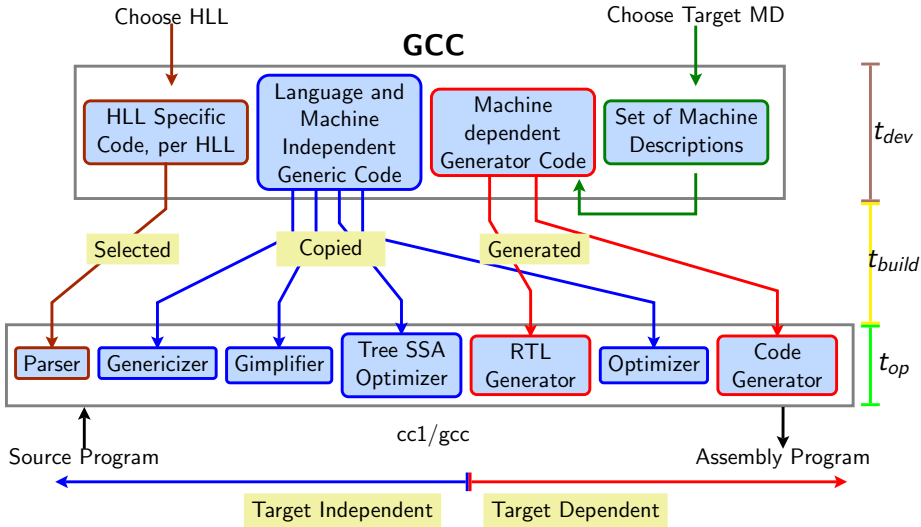
<http://gcc.gnu.org/projects/>

## MORE INFORMATION ...

For more information about GCC

- 1 Main GCC Wiki page  
<http://gcc.gnu.org/wiki>
- 2 Main GCC Internals page  
<http://gcc.gnu.org/onlinedocs/gccint/>
- 3 Main workshop page  
<http://www.cse.iitb.ac.in/~uday/gcc-workshop>  
Downloads section in above has more detailed slides on:
  - 1 GCC Build system
  - 2 Gimple, RTL details
  - 3 Call graph of gcc for various phases
- 4 GCC Internals Documentation by IIT Bombay  
<http://www.cfdvs.iitb.ac.in/~amv/gcc-int-docs>  
Main document: GCC Model.

# THANK YOU



## Part VII

### Additional Points

## Part VIII

# 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

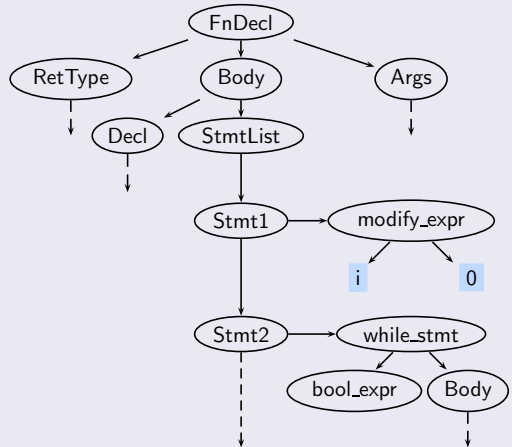
AST = Abstract Syntax  
Tree

Conceptually

Input  
Parse (AST)

Practically ...

Simplified AST



## Conceptually

Input

Parse (AST)

IR<sub>1</sub> (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>;
  ...
}
```

# C Program: Journey through GCC

SSA = Static Single  
Assignment

## Conceptually

Input

Parse (AST)

IR<sub>1</sub> (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<sub>1</sub> (Gimple)
- Optimization
- IR<sub>2</sub> (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<sub>1</sub> (Gimple)  
Optimization  
IR<sub>2</sub> (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
    ...
```

# Note the Transformations

## Observe

- GCC uses three IRs: AST/Generic, Gimple and RTL
- Gimple & AST/Generic – machine independent
- RTL – captures target machine properties
- Transformations are:
  - 1 AST/Generic  $\Rightarrow$  Gimple
  - 2 Gimple  $\Rightarrow$  RTL
  - 3 RTL  $\Rightarrow$  ASM

# GCC Support for “Viewing” the Internals

Phase name	Phase output	Example
Input	–	sample.c
Parse	AST	sample.c.tu
gimplifier	Gimple	sample.c.t03.gimple
Gimple opts	Gimple-cfg etc.	sample.c.t06.useless
RTL Expander	nsRTL	sample.c.00.expand
RTL optimizer	nsRTL	sample.c.08.jump
Reg. Allocator	sRTL	sample.c.37.greg
RTL optimizer	sRTL	sample.c.58.shorten
ASM expander	ASM	sample.s

## Tip

Almost every pass in GCC can output a dump file.

```
-fdump-tree-* -dP
```

## Part IX

# HLLs and Compilation

# Language Abstractions

## Four main abstractions provided by HLLs

- Data Abstractions: Identify and support useful **data** types, and support various aggregation mechanisms
- Control Flow Abstractions: **Control** constructs like sequence, branch, loops, function call etc.
- Procedural Abstractions: Separate **computations** from state and name them
- Name Space Abstractions: Mechanisms to **name** objects in various ways, e.g. by scope rules.

## Key Idea!

Given a pair of languages, identify the abstractions they support and intuitively grasp their “difference”!

# Abstraction Gap between C and i386

## C Abstractions

- 1 Data: char, int, float etc.
- 2 Control: If-then-else branching, finite and infinite iteration loops and function calls.
- 3 Procedural: function calls.
- 4 Naming: statically scoped variables, file scoping.

## i386 Abstractions

- 1 Data: unsigned integers
- 2 Control: Auto-incrementing program counter to support sequencing, and an unrestricted branch.
- 3 Procedural: None (the call instruction does not perform a true procedure call).
- 4 Naming: None.

**Abstraction Gap – an intuitive concept**

Data: smallest gap, & Naming (or procedural): largest gap.

# Compiler: Bridging the Gap

## Why a compiler?

To bridge this gap

## How to do it in a compiler?

Our empirical rule: **Start** handling the largest gap first

Phase Seq 1: name space → procedural → control flow → data.

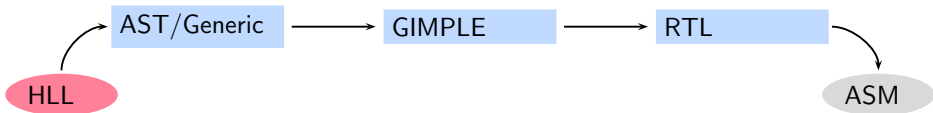
Phase Seq 2: procedural → name space → control flow → data.

## Notes:

- Handling the current gap need not finish before beginning next!
- Phase sequence need not be unique; our rule is empirical!
- GCC uses sequence 2.

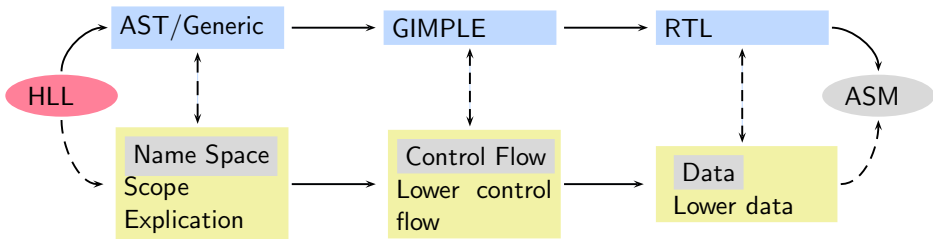
# GCC Phase Sequence and the Abstraction Gap

The GCC Phase Sequence



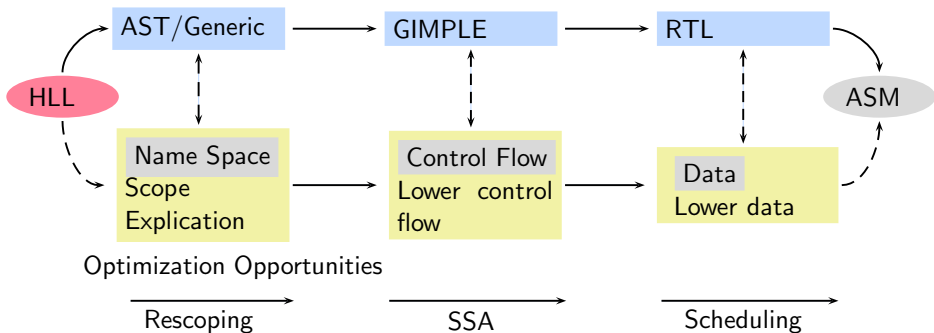
# GCC Phase Sequence and the Abstraction Gap

The GCC Phase Sequence



# GCC Phase Sequence and the Abstraction Gap

The GCC Phase Sequence



## Part X

# About Gimple

# Gimple at $t_{dev}$ : Few GIMPLE Node types

Binary Operator	BIT_AND_EXPR, MAX_EXPR
Comparison	EQ_EXPR, LTGT_EXPR
Constants	INTEGER_CST, STRING_CST
Declaration	FUNCTION_DECL, LABEL_DECL
Expression	MODIFY_EXPR, ADDR_EXPR, BIND_EXPR
Reference	COMPONENT_REF, ARRAY_RANGE_REF
Statement	EXIT_EXPR, GOTO_EXPR
Type	BOOLEAN_TYPE, INTEGER_TYPE
Unary	ABS_EXPR, NEGATE_EXPR, NOP_EXPR

## Tip

- Gimple nodes = AST/Generic nodes - Control flow nodes !
- All tree nodes ( $\sim 152$ ) in GCC in: `$GCCHOME/gcc/tree.def`.
- All nodes types, like above, are enum-med.

A GIMPLE “node” is an instance of:

```
struct tree_common
{
  tree chain;    /* chaining ptr */
  tree type;    /* expression type ptr */
  ...

  /* Node type ($GCCHOME/gcc/tree.def) */
  ENUM_BITFIELD(tree_code) code : 8; /* e.g. MODIFY_EXPR */

  /* Various bit flags */
  unsigned side_effects_flag : 1;
  ...
}; /* $GCCHOME/gcc/tree.h */
```

## Tree Pass Organization

- **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  
};
```

## Part XI

### About RTL

# RTL at $t_{build}$ : The Data Structure for RTL Objects

In rtl.h

```
struct rtx_def {
    /* RTL codes (e.g. SET)  enum-med from
    $GCCHOME/gcc/rtl.def */
    ENUM_BITFIELD(rtx_code)      code : 16;
    ENUM_BITFIELD(machine_mode) mode :  8;
    unsigned int                 jump :  1;
    unsigned int                 unchanging : 1;
    /* ... a few such flags */
    union u {
        rtunion      fld[1];
        HOST_WIDE_INT hwint[1];
    };
};
```

## Part XII

# The GCC Call Graph

# Front End Processing Sequence in cc1 and GCC

```
toplevel_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 */
```

## Tips

1. Use the functions above as breakpoints in gdb on cc1.
2. Source file groups of **entire** phase sequence available.

# GIMPLE Phase sequence in cc1 and GCC

## 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 Optimizations passes */
```

# The Tree passes list

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

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

- Gimple → non-strict RTL translation
- non-strict RTL passes – information extraction & Optimizations
- 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 (`passes.c: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:
  - **Optimize** 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