

# *Retargetability Model of GCC*

Abhijat Vichare

(Contact: amv@cfdvs.iitb.ac.in)

CFDVS,  
Indian Institute of Technology, Bombay



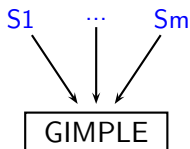
June 2007

# Outline

- GCC MD: Internal issues, the “How” ?
- GIMPLE → RTL: The Retargetting Model of GCC
- Useful Abstractions
- GCC & Cross Compilation



## Recapitulate: The GCC Build

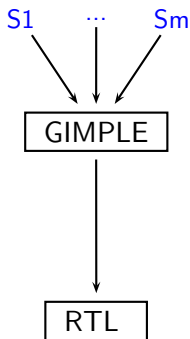


- **Front end:** Multiple source languages
  - ▶ Separate HLL dependent part of code
  - ▶ Selection mechanism required
  - ▶ Parsers for each source
  - ▶ Reduce to a common IR – GIMPLE

GCC Structure



## Recapitulate: The GCC Build

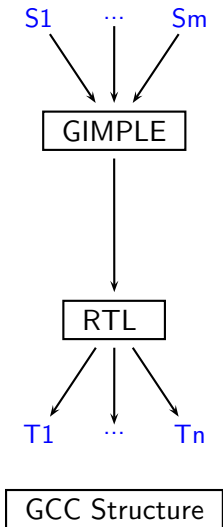


- **Middle:** Optimisations, translations
  - ▶ **Decide:** placement in phase sequence
  - ▶ **Try:** match optimiser needs & IR properties

GCC Structure



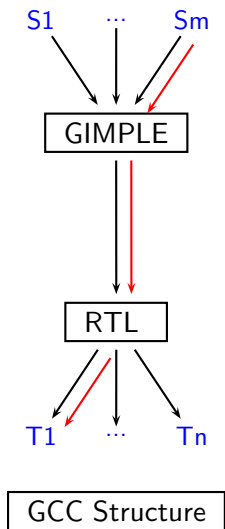
## Recapitulate: The GCC Build



- **Back end:** Multiple targets
  - ▶ Separate target dependent part
  - ▶ Description system for target props
  - ▶ Linear IR preferable



## Recapitulate: The GCC Build



- GCC  $\rightarrow$  gcc/cc1: Build:
  - ▶ Select HLL ( $S_m$ ) & target ( $T_1$ )
  - ▶ Generate target specific code+data  
(gen\*.c < [target].md >  
target-code-or-data)
  - ▶ Build:  $S_m C^{T_1}$ : Compile from  $S_m$  to  $T_1$ .



## What is “Generated”?

- Info about instructions supported by chosen target, e.g.
  - ▶ **Listing** data structures (e.g. instruction pattern lists)
  - ▶ **Indexing** data structures, since diff. targets give diff. lists.
- C functions that **generate** RTL internal representation
- Any useful “attributes”, e.g.
  - ▶ Semantic groupings: arithmetic, logical, I/O etc.
  - ▶ Processor unit usage groups for pipeline utilisation

cc1 = The Complete  $S_m C^{T_1}$  (e.g.  $C C^{SPIM}$ ) compiler, is:

`$GCCHOME/gcc/*. [ch] + $GCCBUILDDIR/gcc/*. [ch]`



## GIMPLE $\rightarrow$ RTL – Again

To translate from GIMPLE to RTL

- **Conceptually:** Associate each GIMPLE node to equivalent RTL expression
- **Implemented:** As a switch-case in the expanders

But

Target independent  $\rightarrow$  target specific + retargetability ?



# Information Required by the CGF

## CGF Compilation Goals

- Utilise the target instruction set as much as possible

## CGF needs

- The **target instructions** – as ASM strings
- Their “**description**” to enable a tailored IR
  
- To **Complete** the GIMPLE → RTL translation tables
- **Mechanisms** to “internalise” the information provided by description
- **Information** about the features of the instructions



# Information Required by the CGF

## CGF Compilation Goals

- Utilise the target instruction set as much as possible

## CGF needs

- The **target instructions** – as ASM strings
- Their “**description**” to enable a tailored IR  
Target tailored IR  $\Rightarrow$  target specific processing without target syntax!
- To **Complete** the GIMPLE  $\rightarrow$  RTL translation tables
- **Mechanisms** to “internalise” the information provided by description
- **Information** about the features of the instructions

Why have a target specific IR ?



## The GIMPLE “Tree” data structure

A GIMPLE “node” is an object of mainly the following:

```
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 */
```



## Information supplied by the MD

- The target instructions – as ASM strings
- A description of the semantics of each
- A description of the features of each like
  - ▶ Data size limits
  - ▶ One of the operands must be a register
  - ▶ Implicit operands
  - ▶ Register restrictions

Information supplied	in <code>define_insn</code> as
The target instruction	ASM string
A description of it's semantics	RTL Template
Operand data size limits	predicates
Register restrictions	constraints



*Part 1*

# *The Retargeting Model of GCC*

## How GCC uses target specific RTL as IR

MODIFY\_EXPR

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



## How GCC uses target specific RTL as IR



## How GCC uses target specific RTL as IR

MODIFY\_EXPR

"movsi"

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

Standard Pattern Name

Separate to CGF code and MD

MODIFY\_EXPR

"movsi"

"movsi"

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



## How GCC uses target specific RTL as IR

MODIFY\_EXPR

"movsi"

(set (&lt;dest&gt;) (&lt;src&gt;))

Standard Pattern Name

Separate to CGF code and MD

MODIFY\_EXPR

"movsi"

"movsi"

(set (&lt;dest&gt;) (&lt;src&gt;))

Implement

MODIFY\_EXPR

"movsi"

"movsi"

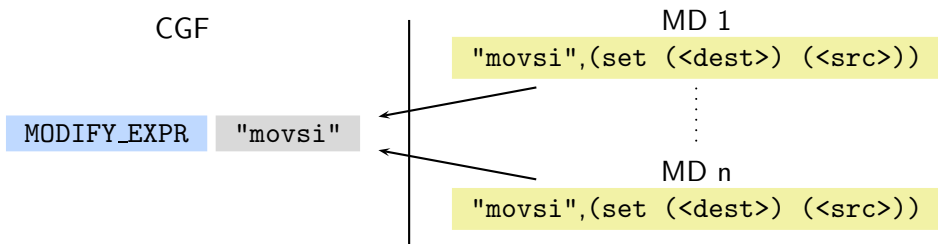
(set (&lt;dest&gt;) (&lt;src&gt;))

Unnecessary in CGF;  
hard code

Implement in MD



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



CGF needs:

An interface **immune** to MD authoring variations



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

CGF

MODIFY\_EXPR

"movsi"

How ?

MD 1

"movsi", (set (&lt;dest&gt;) (&lt;src&gt;))

⋮

MD n

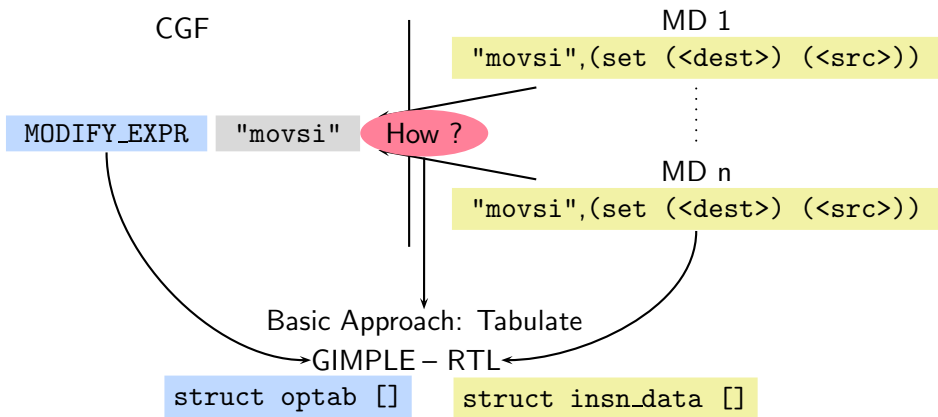
"movsi", (set (&lt;dest&gt;) (&lt;src&gt;))

CGF needs:

An interface **immune** to MD authoring variations



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



### CGF needs:

An interface **immune** to MD authoring variations



## MD Authoring Immune Tabulation

- List insns as they appear in the chosen MD
- Index them
- Supply index to the CGF

### Note

An SPN may be written at any suitable place for a given MD



# MD Information Data Structures

## Two principal data structures

- `struct optab` – Interface to CGF
- `struct insn_data` – All information about a pattern
  - ▶ Array of each pattern read
  - ▶ Some patterns are SPNs
  - ▶ Each pattern is accessed using the generated index

## Supporting data structures

- `enum insn_code`: Index of patterns available in the given MD

## Note

Data structures are named in the CGF, but populated at build time.  
Generating target specific code = populating these data structures.



## GCC Generation Phase – Revisited

Generator	Generated from MD	Information	Description
genopinit	insn-opinit.c	void init_all_optabs (void);	Operations Table Initialiser
gencodes	insn-codes.h	enum insn_code = { ... CODE_FOR_movsi = 23, ... }	Index of patterns
genooutput	insn-output.c	struct insn_data [CODE].genfun = /* fn ptr */	All insn data e.g. gen function
genemit	insn-emit.c	rtx gen_rtx_movsi (/* args */ { /* body */ }	RTL emission functions



## RTL Generation – The Internals

```
case MODIFY_EXPR: ... expand_assignment (...);
    ... /* Various cases of expansion */
/* One case: integer mode move */
icode = mov_optab->handler[SImode].insn_code
if (icode != CODE_FOR_nothing) {
    ... /* preparatory code */
    emit_insn (GEN_FCN(icode)(dest,src));
}
```



## RTL Generation – The Internals

TREE node

```
case MODIFY_EXPR: ... expand_assignment (...);
    ... /* Various cases of expansion */
/* One case: integer mode move */
icode = mov_optab->handler[SImode].insn_code
if (icode != CODE_FOR_nothing) {
    ... /* preparatory code */
    emit_insn (GEN_FCN(icode)(dest,src));
}
```



## RTL Generation – The Internals

Seek index

```
case MODIFY_EXPR: ... expand_assignment (...);
    ... /* Various cases of expansion */
/* One case: integer mode move */
icode = mov_optab->handler[SImode].insn_code
if (icode != CODE_FOR_nothing) {
    ... /* preparatory code */
    emit_insn (GEN_FCN(icode)(dest,src));
}
```



## RTL Generation – The Interr

```
case MODIFY_EXPR: ... expand_assignment (
    ... /* Various cases of expansion */
    /* One case: integer mode move */
    icode = mov_optab->handler[SImode].insn_code
    if (icode != CODE_FOR_nothing) {
        ... /* preparatory code */
        emit_insn (GEN_FCN(icode)(dest,src));
    }
```

insn-codes.h

```
enum insn_code
= { ...
  CODE_FOR_movsi
= 23,
  ... }
```



## RTL Generation – The Intern

```

case MODIFY_EXPR: ... expand_assignment (
    ... /* Various cases of expansion */
    /* One case: integer mode move */
    icode = mov_optab->handler[SImode].insn_code
    if (icode != CODE_FOR_nothing) {
        ... /* preparatory code */
        emit_insn (GEN_FCN(icode)(dest,src));
    }

```

```

insn-codes.h
enum insn_code
= { ...
  CODE_FOR_movsi
= 23,
  ... }

```

Got index into optab



## RTL Generation – The Internals

```
case MODIFY_EXPR: ... expand_assignment (...);
    ... /* Various cases of expansion */
/* One case: integer mode move */
icode = mov_optab->handler[SImode].insn_code
if (icode != CODE_FOR_nothing) {
    ... /* preparatory code */
    emit_insn (GEN_FCN(icode)(dest,src));
}
```

Use icode (= 23)

```
#define GENFCN(code) insn_data[code].genfun
```



## RTL Generation – The Internals

```
case MODIFY_EXPR: ... expand_assignment (...);
    ... /* Various cases of expansion */
/* One case: integer mode move */
icode = mov_optab->handler[SImode].insn_code
if (icode != CODE_FOR_nothing) {
    ... /* preparatory code */
    emit_insn (GEN_FCN(icode)(dest,
}
```

```
insn-output.c
insn_data[23].genfun
= gen_rtx_movsi
```

```
#define GENFCN(code) insn_data[code].genfun
```

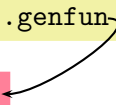


## RTL Generation – The Internals

```
case MODIFY_EXPR: ... expand_assignment (...);
    ... /* Various cases of expansion */
/* One case: integer mode move */
icode = mov_optab->handler[SImode].insn_code
if (icode != CODE_FOR_nothing) {
    ... /* preparatory code */
    emit_insn (GEN_FCN(icode)(dest,src));
}
```

```
#define GENFCN(code) insn_data[code].genfun
```

```
Execute: gen_rtx_movsi (dest,src)
```



## Conversion of RTL from Textual to Internal Form

```
(define_expand "movsi"  
  [(set (op0) (op1))]  
  ""  
  "{ /* C CODE OF DEFINE EXPAND */ }")
```



## Conversion of RTL from Textual to Internal Form

```
(define_expand "movsi"  
  [(set (op0) (op1))]  
  ""  
  "{ /* C CODE OF DEFINE EXPAND */ }")
```

```
rtx  
gen_movsi (rtx operand0, rtx operand1)  
{  
  ...  
  {  
    /* C CODE OF DEFINE EXPAND */  
  }  
  emit_insn (gen_rtx_SET (VOIDmode, operand0, operand1))  
  ...  
}
```



## Conversion of RTL from Textual to Internal Form

```
(define_expand "movsi"  
  [(set (op0) (op1))]  
  ""  
  "{ /* C CODE OF DEFINE EXPAND */ }")  
  
rtx  
gen_movsi (rtx operand0, rtx operand1)  
{  
  ...  
  {  
    /* C CODE OF DEFINE EXPAND */  
  }  
  emit_insn (gen_rtx_SET (VOIDmode, operand0, operand1))  
  ...  
}
```

The diagram illustrates the conversion of RTL from textual to internal form. It shows two versions of the `define_expand` macro for `"movsi"`. The top version is the textual form, and the bottom version is the internal form. Arrows indicate the mapping between the two forms: the `set` token in the textual form is mapped to the `SET` token in the internal form, and the C code comment `/* C CODE OF DEFINE EXPAND */` in the textual form is mapped to the same comment in the internal form.



## Conversion of RTL from Textual to Internal Form

```
(define_expand "movsi"  
  [(set (op0) (op1))]  
  ""  
  "{ /* C CODE OF DEFINE EXPAND */ }")
```

```
rtx  
gen_movsi (rtx operand0, rtx operand1)  
{  
  ...  
  {  
    /* C CODE OF DEFINE EXPAND */  
  }  
  emit_insn (gen_rtx_SET (VOIDmode, operand0, operand1))  
  ...  
}
```

IR object to create



## Conversion of RTL from Textual to Internal Form

```
(define_expand "movsi"  
  [(set (op0) (op1))]  
  ""  
  "{ /* C CODE OF DEFINE EXPAND */ }")
```

```
rtx  
gen_movsi (rtx operand0, rtx operand1)  
{  
  ...  
  {  
    /* C CODE OF DEFINE EXPAND */  
  }  
  emit_insn (gen_rtx_SET (VOIDmode, operand0, operand1))  
  ...  
}
```

Create IR object



## Conversion of RTL from Textual to Internal Form

```
(define_expand "movsi"  
  [(set (op0) (op1))]  
  ""  
  "{ /* C CODE OF DEFINE EXPAND */ }")
```

```
rtx  
gen_movsi (rtx operand0, rtx operand1)  
{  
  ...  
  {  
    /* C CODE OF DEFINE EXPAND */  
  }  
  emit_insn (gen_rtx_SET (VOIDmode  
  ...  
}
```

Chain the created  
IR object into program IR



## The RTX data structure

In rtl.h

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



*Part 2*

*Useful Abstractions*

## GCC IRs as Abstract Machines

Because GIMPLE and RTL can represent any computation, we can view them as abstract machines.

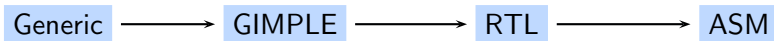
- GIMPLE: Symbolic, heap based memory; low level control flow, implicit “program counter” – the tree walker; scope chain resolved
- RTL: Numeric, linear memory (pseudo registers and memory), low level data and operators, implicit “program counter” – the `insn` chain pointer; scope references resolved.

With this insight, the `cc1` phase sequence looks like ...



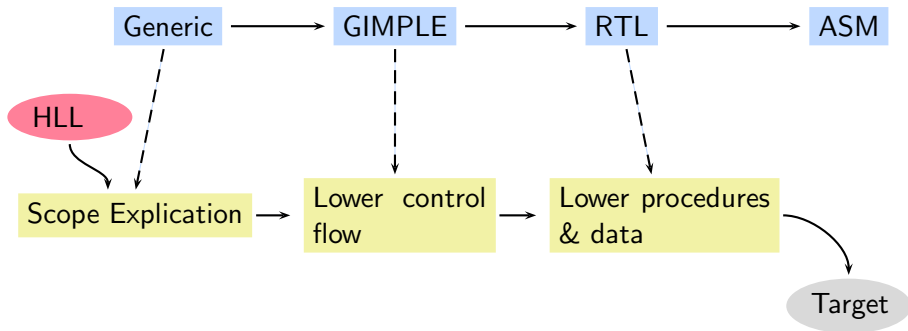
# The cc1 Phase Sequence as IR Chain

The GCC Phase Sequence



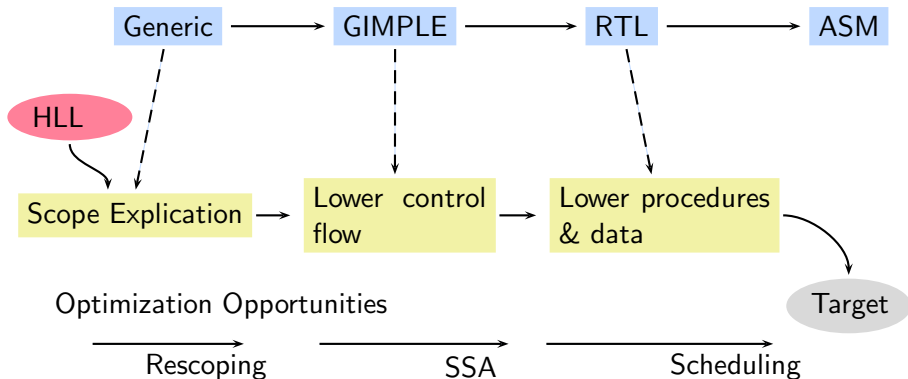
# The cc1 Phase Sequence as IR Chain

## The GCC Phase Sequence



# The cc1 Phase Sequence as IR Chain

## The GCC Phase Sequence



*Part 3*

# *GCC & Cross Compilation*

## GCC & Cross Compilation

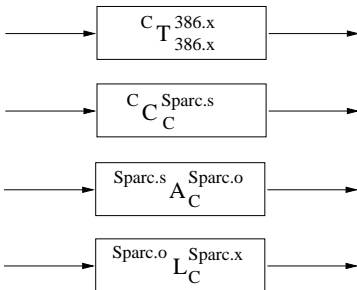
Because GCC is retargetable, one can use it as the cross compiler component of a complete cross development system.

In the following slides we will pictorially describe the construction of a complete cross tool chain.



# The Cross Tool Chain Problem

Given:

 $C_{LIB}$ 

Notation:

T: Complete translation system

C: Compiler (source to assembly)

A: Assembler (assembly to object)

L: Linker (object to executable)

LIB: Library of objects

Left superscript: "Language" of input

Right superscript: "Language" of output

Right subscript: "Language" of execution

Suffixes: .s : assembly source

.o : object

.x : executable

→ : Infix sequence operation

|| : Infix parallel operation

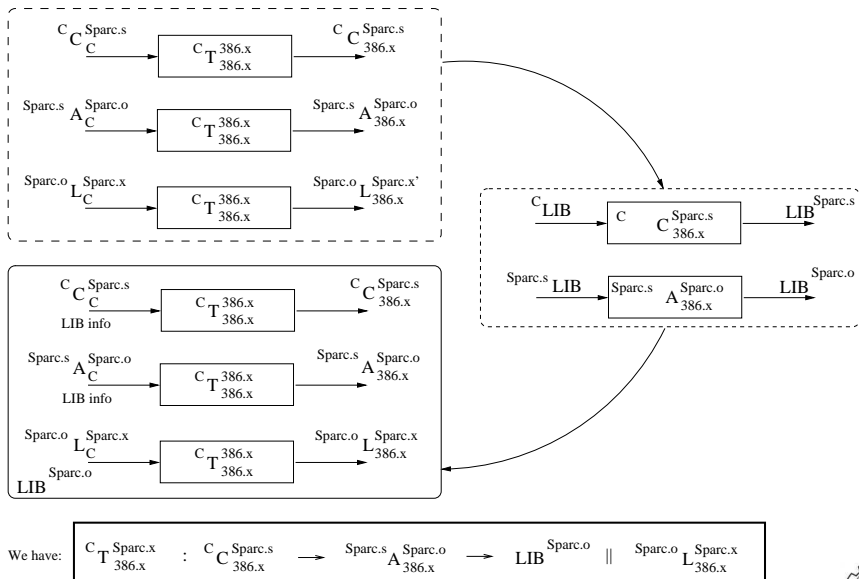
where:

$$C_{T_{386.x}^{386.x}} : C_{C_{386.x}^{386.s}} \rightarrow^{386.s} A_{386.x}^{386.o} \rightarrow LIB^{386.o} \parallel^{386.o} L_{386.x}^{386.o}$$

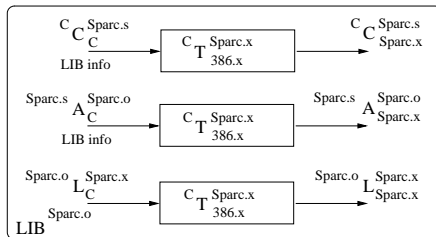
Problem: Generate:

$$C_{T_{Sparc.x}^{Sparc.x}}$$


# The Cross Tool Chain Problem



# The Cross Tool Chain Problem



We have:

$$C_T \text{ Sparc.x} \text{ Sparc.x} : C_C \text{ Sparc.s} \text{ Sparc.x} \rightarrow \text{Sparc.s} \text{ A Sparc.o} \text{ Sparc.x} \rightarrow \text{LIB Sparc.o} \parallel \text{Sparc.o} \text{ L Sparc.x} \text{ Sparc.x}$$



*Part 4*

*Summary*

## Summary

- The GCC CGF uses the MD information to complete the GIMPLE → RTL translation table.
- This is done at **build** time.
- The target specific information is **first** collected into data structures.
- The CGF and the target specific part are then built to generate the desired  $S_m C^{T_1}$  compiler.
- Viewing the GCC IRs as abstract machines is useful to understand the overall phase sequence of the compiler.
- As a retargetable framework, GCC offers an opportunity to construct cross development tool chain.



THANK YOU