GCC Internals & Code Generation

Mukkaysh Srivastav

srimks@msn.com

August 5, 2009

Demystifying GCC

What is GCC?

Why use GCC?

What does compilation with GCC look like?

About this presentation

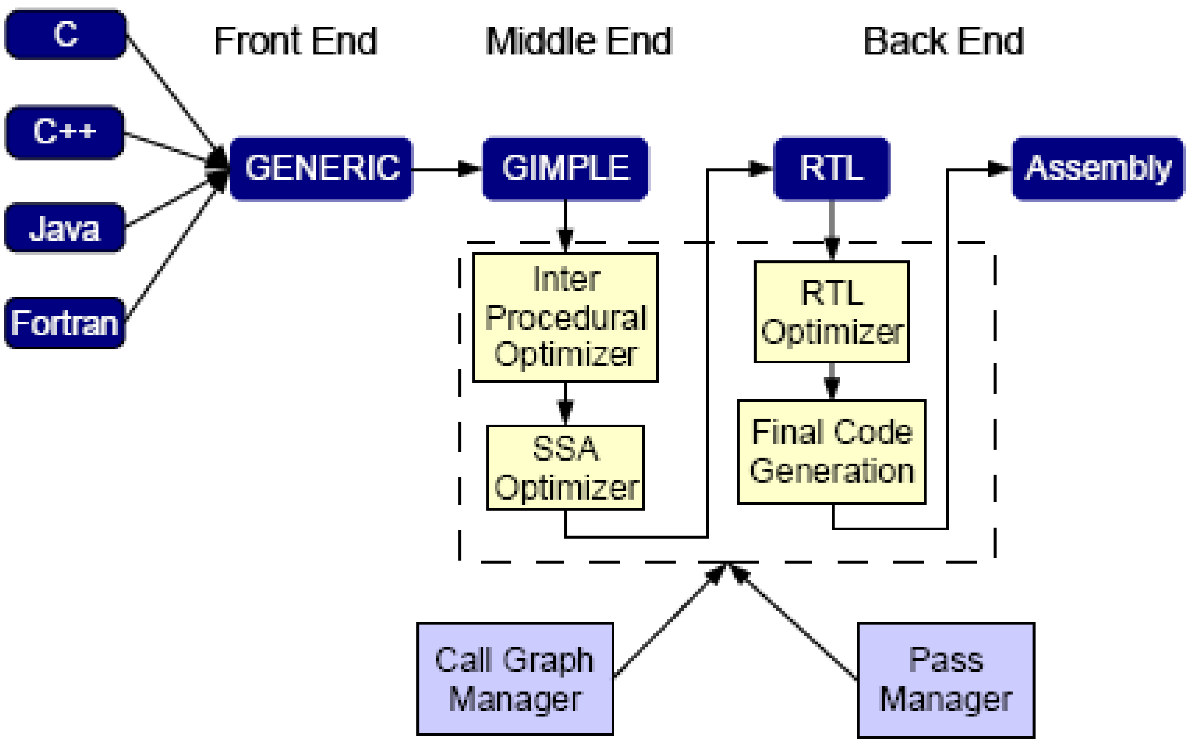
The discussion intends to focus on

- Compilers for embedded system
  - Requirement of embedded compiler
  - How GCC fits in?
- Still, why understand working of GCC?
- Introduction to GCC backend
  - i.e. Concentrate mainly on GCC machine description (*.md) file
  - Examples on how to modify and use machine descriptor file

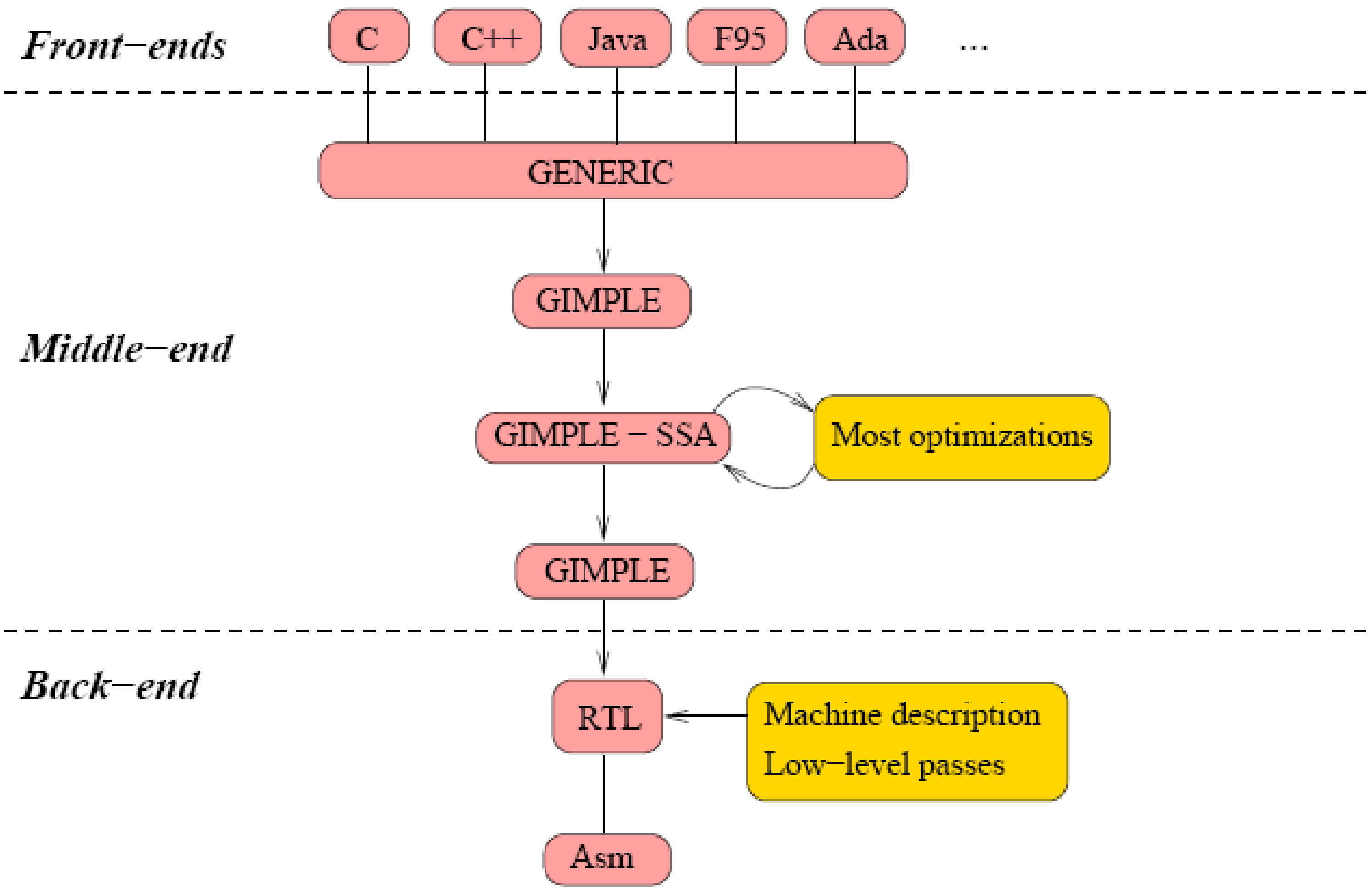
Overall why having open source compiler gives added advantage in embedded system development.

Contents

GCC ....Discussion Overview



Compiler Structure



GCC Overview

- Free Software (GPL), Open & Distributed Development Processes, Deeply embedded to platforms, Supports major languages – C, C++, Java, Fortran 95, Ada, Obj C/C++, etc.

- GCC a machine-independent code but dependent on machine-parameters such as – endianess and availability of auto-increment addressing.

- Bootstraps on native platforms, Warning-free, Extensive Regression-suite, Peer review by maintainers, Strict coding standard & Patch revision policy.

- SSA-based high-level global optimizer, Constraint-based points-to alias analysis, Data dependency based on chain of recurrences, Feedback directed optimizations, Interprocedural Optimization, Automatic pointer checking instrumentation, Automatic loop vectorization, OpenMP support.

GCC ....Embedded System Compiler

Qualities for Embedded System Compilers

- Support wide variety of processor.
- Be a good Optimizing compiler
- Allow different target ABI
- Easily Reconfigurable & full tool chain support
- Inexpensive

How about GCC ?

- Work with any and all type of target processors (RISC/CISC, Little Endian/Big Endian etc)
- Is a good optimizing compiler
- Allow target ABI of 8, 16, 32, 64 bit int type.
- Support for C, CPP & other languages, provide all Debug related data using DDD/GDB debugger
- Its Free!!!!

Compilers - Basics

- **Lexical Analysis -** The lexical analyzer reads the source program and emits tokens. Tokens are atomic units, which represent a sequence of characters. They can be treated as single logical entitities. Identifiers, keywords, constants, operators, punctuation symbols etc. are examples of tokens.

- **Syntax Analysis -** Tokens from the lexical analyzer are the input to this phase. A set of rules defines the grammar of the language. The syntax analyzer checks whether the given input is a valid one. i.e.. whether it is permitted by the given grammar.

- **Immediate Code-Generation -** Once the syntactic constructs are determined, the compiler can generate object code for each construct. But the compiler creates an intermediate form called parse tree. A parse tree may contain variables as the terminal nodes. A binary operator will be having a left and right branch for operand1 and operand2.

- **Code-Optimization -** Optimization involves the technique of improving the object code created from the source program.

- Code-Generation - The code generation phase converts the intermediate code generated into a sequence of machine instructions

- Symbols - A data structure used for collecting the variables names is known as a symbol table.

Compiler Tools

- **Flex -** Flex is a *fast lexical analyzer generator*. The first phase of building a compiler is lexical analysis. Input file will have an extension .l, which shows that it is a valid lex file. The output of the flex is a file called lex.yy.c. It has a routine yylex() defined in it. The file, lex.yy.c can be compiled and linked with the '-lfl' library to produce the executable.

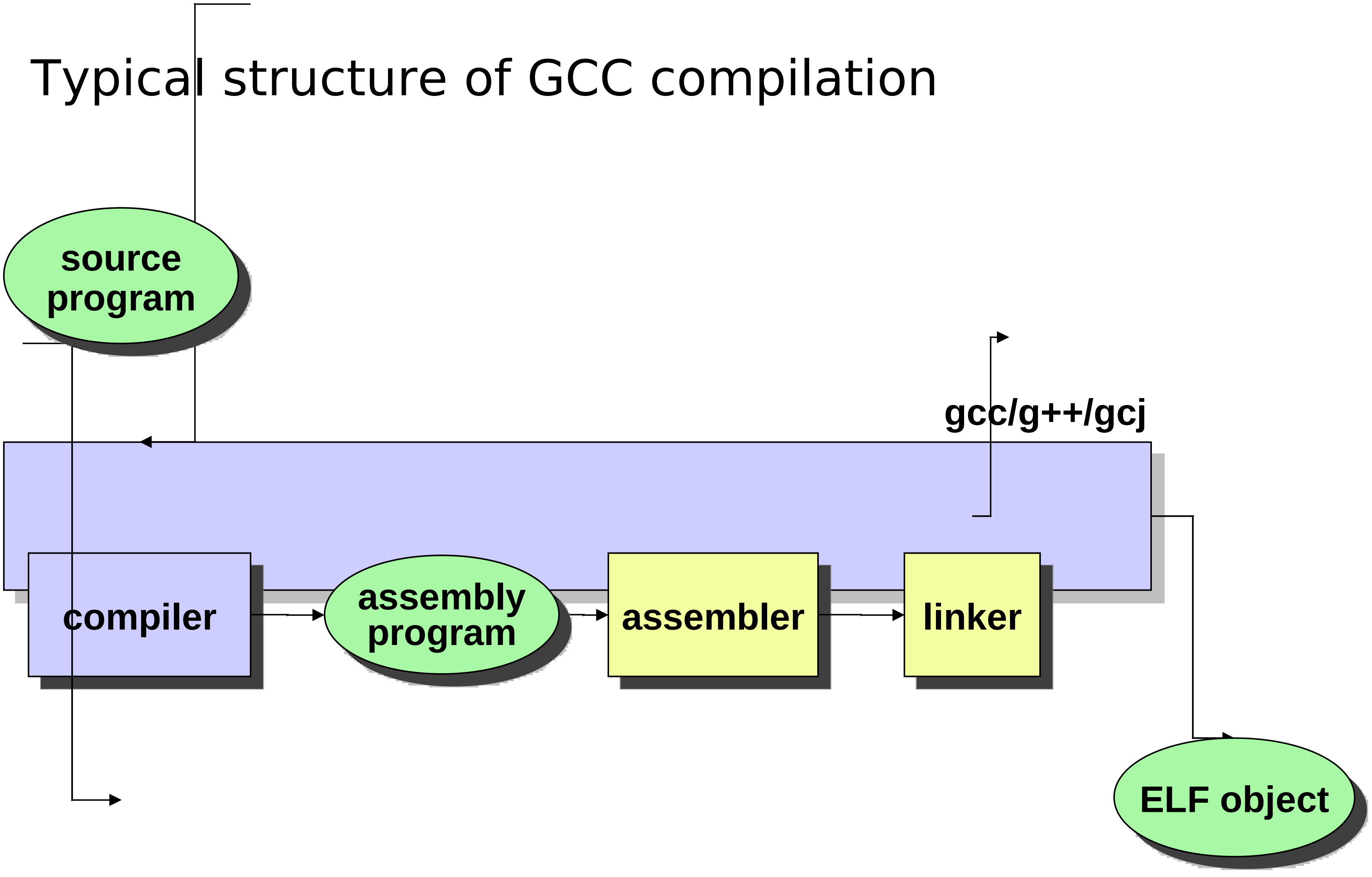
- **Bison –** *Bison* is a parser generator. Given a context-free grammar, it is the duty of Bison to generate a C program to parse that grammar.

- **Context Free Grammar –** Any grammar expressed in BNF is a *context-free grammar*. CFG has -
  - a set of *terminal symbols*, which are the characters of the alphabet that appear in the strings generated by the grammar.
  - a set of *nonterminal symbols*, which are placeholders for patterns of terminal symbols that can be generated by the nonterminal symbols.
  - a set of *productions*, which are rules for replacing (or rewriting) nonterminal symbols (on the left side of the production) in a string with other nonterminal or terminal symbols (on the right side of the production).
  - a *start symbol*, which is a special nonterminal symbol that appears in the initial string generated by the grammar.

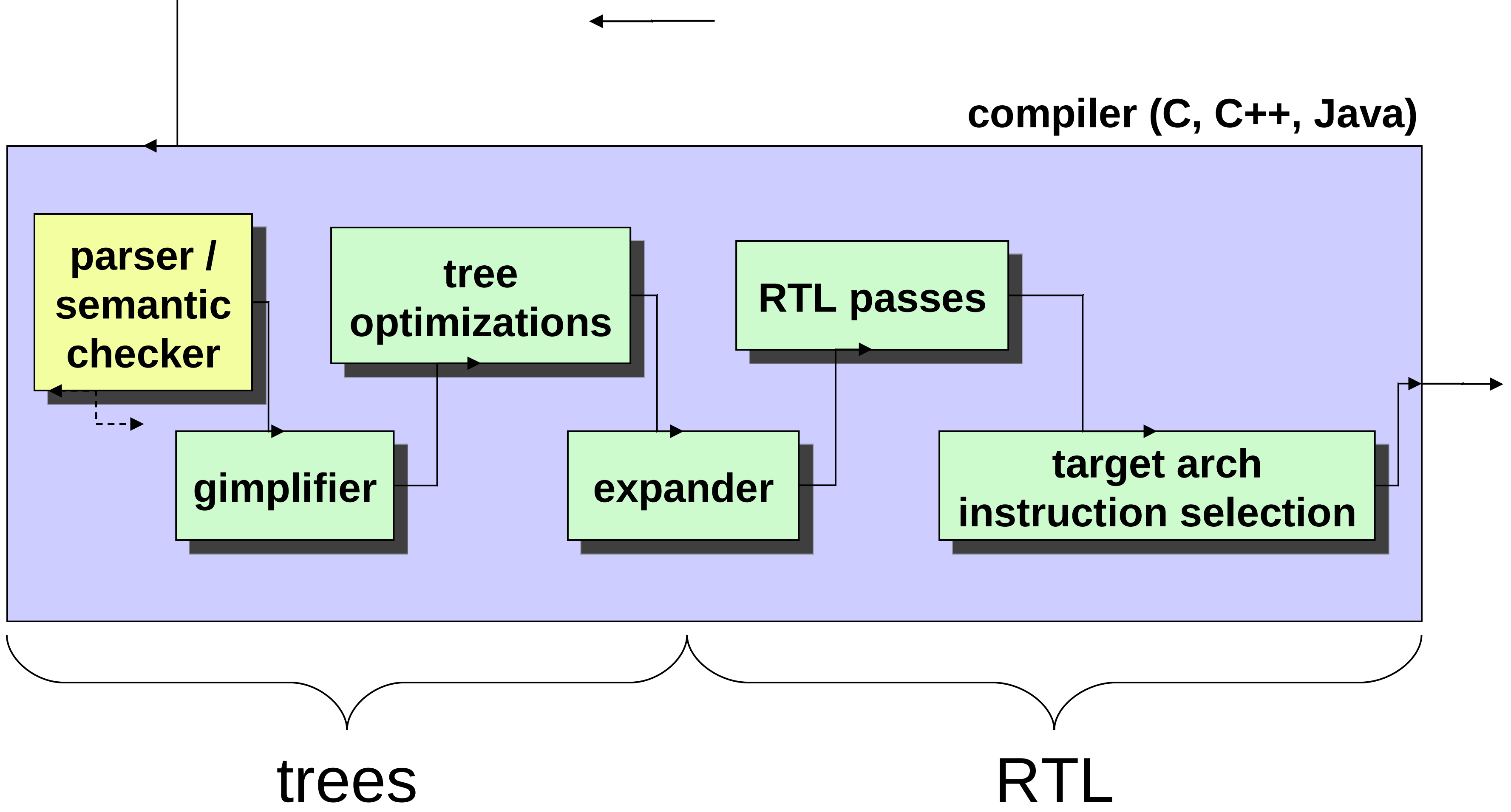
- **Parsing -** Shift-Reduce (SR), Left-Right (LR), Top-DOWN Parsing (Recursive-descent), Bottom-UP Parsing, LL(1), LR(0), SLR(1), LALR(1), LR(1).

GCC Compilation

Typical structure of GCC compilation



Inside the compiler



GCC Compilation Process Structure

GCC-based compiler can be conceptually split into three phases:

- Front-End: A *front end* takes the source code, and does whatever is needed to translate that source code into a semantically equivalent, language independent abstract syntax tree (AST). The syntax and semantics of this AST are defined by the GIMPLE language, the highest level language independent intermediate representation GCC has.

- Middle-End: This AST is then run through a list of target independent code transformations that take care of such things as constructing a control flow graph, and optimizing the AST for optimizing compilations, lowering to non-strict RTL(expand), and running RTL based optimizations for optimizing compilations. The non-strict RTL is handed over to more low-level passes.

- Back-End: The low-level passes are the passes that are part of the code generation process.
  - The first job of these passes is to turn the non-strict RTL representation into strict RTL, or in other words, from RTL patterns that match define_insn definitions without taking constraints into consideration into RTL patterns that fully match the complete insn definition including all operand constraints. Other jobs of the strict RTL passes include scheduling, doing peephole optimizations, and emitting the assembly output.

Demystifying GCC

GCC Basics

Part I

How do you build GCC?

How do you navigate the source tree?

How to differentiate – build, host & target?

GCC Basics: Getting Started

- Requirements to build GCC
  - usual suite of UNIX tools (C compiler, assembler/linker, GNU Make, tar, awk, POSIX shell)
- For development
  - GNU m4 and GNU autotools (autoconf/automake/libtool)
  - gperf
  - bison, flex
  - autogen, guile, gettext, perl, Texinfo, diffutils, patch, …
- Obtaining GCC sources
  - gcc.gnu.org or local mirror (see gcc.gnu.org/mirrors.html)
  - get gcc-core package, then language add-ons
    - gcc-java requires gcc-g++

Building GCC from sources

- Configure it in a *separate* build directory from sources
  - **/path/to/source/directory/configure *options…***
  - --prefix=*install-location*
  - --enable-languages=*comma-separated-language-list*
  - --enable-checking
    - turns on sanity checks (especially on intermediate representation)
- Build it ***!***
  - Environment variables useful when debugging compiler/runtime
    - CFLAGS stage 1 flags (using host C compiler)
    - BOOT_CFLAGS stage 2 and stage 3 flags (using stage 1 GCC)
    - CFLAGS_FOR_TARGET flags for new GCC building target binaries
    - CXXFLAGS_FOR_TARGET
    - flags for new GCC building libstdc++/others
    - GCJFLAGS flags for new GCC building Java runtime
    - '**-O0 –ggdb3**' is recommended when debugging

Building GCC from sources

- Build it *!* *continued…*
  - **`make bootstrap`** `(to bootstrap)` *`or`* **`make`** `(to not)`
    - *bootstrap* useful when compiling with non-GCC host compiler
    - during development, non-bootstrap is faster and also better at recompiling just those sources that have changed
  - use make's -j option to speed things up on MP/dual core
  - `make bootstrap-lean`
    - cleans up between stages, uses less disk
  - `make profiledbootstrap`
    - faster compiler produced, but need GCC host
    - –j unsupported
- Install it *!*
  - `make install`

Building a cross-compiler

- Code generator can be built for any target
  - runtime libraries then are built using that code generator

- Since GCC outputs assembly, you actually need a full cross development toolchain
  - Dan Kegel’s crosstool automates a GNU/Linux cross chain for popular configurations:
    - Linux kernel headers
    - GNU binutils
    - glibc
    - gcc
    - **see kegel.com/crosstool**

GCC Basics: Getting Around

- Other tools recommended when hacking GCC

•GNU Screen	attach/reattach terminal sessions
•etags	navigation to source definitions (emacs)
•ctags	navigation to source definitions (vi)
•c++filt	demangle C++/Java mangled symbols
•readelf	decompose ELF files
•objdump	object file dumper/disassembler
•gdb	GNU debugger

GCC Drivers

- gcc, g++, gcj are *drivers*, not *compilers*
  - They will execute (as appropriate):
    - compiler (cc1, cc1plus, jc1)
    - Java program main entry point generation (jvgenmain)
    - assembler (as)
    - linker (collect2)

- Differences between drivers include active #defines, default libraries, other behavior
  - but can use any driver for any source language

Most useful driver options for debugging

• **`-E`**	preprocess, don’t compile
• **`-S`**	compile, don’t assemble
• **`-H`**	verbose header inclusion
• **`-save-temps`**	save temporary files
• **`-print-search-dirs`**	print search paths
• **`-v`**	verbose (see what the driver does)
• **`-g`**	include debugging symbols

• **--help**	get command line help
• **--version**	show full version info
• **-dumpversion**	show minimal version info

GCC source Tour

•INSTALL	configuration/installation documentation
•boehm-gc	the Boehm garbage collector
•config	architecture-specific configure fragments
•contrib	contributed scripts
•fastjar	a replacement for the jar tool
•fixincludes	source for a program to fix host header files when they aren't ANSI-compliant
•gcc	the main compiler source
•include	headers used by GCC (libiberty mostly)
•intl	support for languages other than English

GCC source Tour ...contd.

• libada	ada runtime library
• libcpp	source for C preprocessing library
• libffi	Foreign Function Interface library (allows function callers and receivers to have different calling conventions)
• libiberty	useful utility routines (symbol tables *etc.*) used by GCC and replacement functions for common things not provided by host
• libjava	source for standard Java library
• libmudflap	source for a pointer instrumentation library
• libstdc++-v3	source for standard C++ library
• maintainer-scripts	utility scripts for GCC maintainers
• zlib	compression library source

GCC Source Code

- GCC top level sub-directory

<src>

gcc	Front/middle/back ends
libgcc	Internal library for missing target features
libcpp	Pre-processor
libada	Ada runtime
libstdc++-v3	C++ runtime
libgfortran	Fortran runtime
libobjc	Objective-C runtime
boehm-gc libffi libjava zlib	Java runtime
libiberty	Utility functions and generic data structures
libgomp	OpenMP runtime
libssp	Stack Smash Protection runtime
libmudflap	Pointer/memory check runtime
libdecnumber	Decimal arithmetic library

GCC Source Code ....contd.

- gcc sub-directory

<src>

gcc	Core compiler and C front end
ada	Ada front end
config	Architecture-specific codegen (ports)
cp	C++ front end
doc	User manual and internal documentation
fortran	Fortran front end
ginclude	System headers (mainly freestanding support)
java	Java front end
objc	Objective C front end
objcp	Objective C++ front end
po	Portable object files for I18N
testsuite	Regression tests
treelang	Toy language front end

GCC Source Code ....contd.

Core compiler files (<src>/gcc)

- Alias analysis
- Build support
- C front end
- CFG and callgraph
- Code generation
- Diagnostics
- Driver
- Profiling
- Internal data structures
- Mudflap
- OpenMP
- Option handling
- RTL optimizations
- Tree SSA optimizations

GCC Source Code .. Configuring & Building

```
$ svn co svn://gcc.gnu.org/svn/gcc/trunk
$ mkdir bld && cd bld
$ ../trunk/configure --prefix=`pwd`
$ make all install
```

- Bootstrap is a 3 stage process
  - Stage 0 (host) compiler builds Stage 1 compiler
  - Stage 1 compiler builds Stage 2 compiler
  - Stage 2 compiler builds Stage 3 compiler
  - Stage 2 and Stage 3 compilers must be binary identical

GCC Source Code ....Common Configuration Options

--prefix

- Installation root directory

--enable-languages

- Comma-separated list of language front ends to build
- Possible values

  `ada,c,c++,fortran,java,objc,obj-c++,treelang`

- Default values

  `c,c++,fortran,java,objc`

--disable-bootstrap

- Build stage 1 compiler only

--target

- Specify target architecture for building a cross-compiler
- Target specification form is (roughly)

  ```
  cpu-manufacturer-os
  cpu-manufacturer-kernel-os
  ```

  e.g.

  ```
  x86_64-unknown-linux-gnu
  arm-unknown-elf
  ```

- All possible values in `<src>/config.sub`

GCC Source Code ....Common Configuration Options

`--enable-checking=list`

- Perform compile-time consistency checks
- List of checks: `assert fold gc gcac misc rtl rtlflag runtime tree valgrind`
- Global values:

`yes`	→	`assert,misc,tree,gc,rtlflag,runtime`
`no`	→	Same as `--disable-checking`
`release`	→	Cheap checks `assert,runtime`
`all`	→	Everything except `valgrind`

SLOW!

GCC Source Code .... Common Build Options

- all
  - Default make targets. Knows whether to bootstrap or not
- install
  - Not necessary but useful to test installed compiler
  - Set LD_LIBRARY_PATH afterward
- check
  - Run the test-suite. Use with –k to prevent stopping from when some tests fails
- clean
  - That, and all the other files built by “make all”
- distclean
  - That, and all the other files created by “configure”
- uninstall
  - deletes installed files

GCC Source Code .... Build Results

- Staged compiler binaries

❶ `<bld>/stage1-{gcc,intl,libcpp,libdecnumber,libiberty}`

❷ `<bld>/prev-{gcc,intl,libcpp,libdecnumber,libiberty}`

❸ `<bld>/{gcc,intl,libcpp,libdecnumber,libiberty}`

- Runtime libraries are not staged, except `libgcc`

  `<bld>/<target-triplet>/lib*`

- Testsuite results

  `<bld>/gcc/testsuite/*.{log,sum}`

  `<bld>/<target-triplet>/lib*/testsuite/*.{log,sum}`

- Compiler is split in several binaries

`<bld>/gcc/xgcc`	Main driver
`<bld>/gcc/cc1`	C compiler
`<bld>/gcc/cc1plus`	C++ compiler
`<bld>/gcc/jc1`	Java compiler
`<bld>/gcc/f951`	Fortran compiler
`<bld>/gcc/gnat1`	Ada compiler

- Main driver forks one of the `*1` binaries
- `<bld>/gcc/xgcc -v` shows what compiler is used

GCC Source Code .... Analyzing Test Results

- The best way is to have two trees built
  - pristine
  - pristine + patch
- Pristine tree can be recreated with

```
$ cp -a trunk trunk.pristine
$ cd trunk.pristine
$ svn revert -R .
```

- Configure and build both compilers with the exact same flags
- Use `<src>/trunk/contrib/compare_tests` to compare individual `.sum` files

```
$ cd <bld>/gcc/testsuite/gcc

$ compare_tests <bld.pristine>/gcc/testsuite/gcc/gcc.sum gcc.sum

   Tests that now fail, but worked before:
   gcc.c-torture/compile/20000403-2.c  -Os  (test for excess errors)

   Tests that now work, but didn't before:
   gcc.c-torture/compile/20000120-2.c  -O0  (test for excess errors)
   gcc.c-torture/compile/20000405-2.c  -Os  (test for excess errors)
```

GCC Source Code .... Patch Submission

- Non-trivial contributions require copyright assignment
- Code should follow the GNU coding conventions
  - http://www.gnu.org/prep/standards_toc.html
  - http://gcc.gnu.org/codingconventions.html
- Submission should include
  - `ChangeLog` describing what changed (not how nor why)
  - Test case (if applicable)
  - Patch itself generated with `svn diff` (context or unified)
- When testing a patch
  1. Disable bootstrap
  2. Build C front end only
  3. Run regression testsuite
  4. Once all failures have been fixed
     - Enable all languages
     - Run regression testsuite again
  5. Enable bootstrap
  6. Run regression testsuite

Not strictly necessary, but recommended

- Patches are only accepted after #5 and #6 work

GCC Testsuite

- C Language DejaGnu testsuite
  - gcc.dg/compat – test for binary compatibility using ‘compat-exp’
  - gcc.dg/cpp – tests of the preprocessor
  - gcc.dg/debug – tests for debug formats
  - gcc.dg/format – tests of the ‘-Wformat’ format checking
  - gcc.dg/non-compile - contains test of codes that should not compile & should not need any special compilations options.
  - gcc.dg/special – FIXME: describe this
  - gcc.c-torture/compat - FIXME: describe this
  - gcc.c-torture/compile – testcases that should compile, but do not need to link or run
  - gcc.c-torture/execute – testcases that should compile, link & run.
  - gcc.c-torture/execute/ieee – tests that are specific to IEEE floating point
  - gcc.c-torture/unsorted - FIXME: describe this
  - gcc.c-torture/misc-tests – tests that requires special handling

Demystifying GCC

Front-end Middle-end Back-end

The GCC Front-End

Part II

Option processing
Controlling drivers and hooking up front-ends
The C and C++ front-ends
The GENERIC high-level intermediate representation

GCC Front-End ....Advantages

- The objective of the front end is to read the source file, parse it, and convert it into the standard *abstract syntax tree (AST)* representation.

- The AST is a dual-type representation: it is a tree where a node can have children and a list of statements where nodes are chained one after another. There is one front end for each programming language.

- The AST is then used to generate a *register-transfer language (RTL)* tree. RTL is a hardware-based representation that corresponds to an abstract target architecture with an infinite number of registers.

- GCC Front-End benefit from the support of many different targets machines already present in GCC.

- GCC Front-End benefits from all the optimizations in the GCC.

- Better debugging information is generated when compiling directly from source code than when going via intermediate generated code.

- Code-reusability

GCC Front-End ...contd.

- gcc, g++, gcj driver entry point
  - main (gcc/gcc.c)
- cc1, cc1plus, jc1 share a common entry point
  - toplev_main (gcc/toplev.c)
    - actual main in gcc/main.c
      - just calls toplev_main()
      - can be overridden by front-end

- In gcc/ directory

common.opt	option definitions
opts.{c,h}	**common_handle_option()**
c-opts.c	**c_common_handle_option()**
c.opt	C compiler option definitions
java/lang.opt	Java compiler option definitions
java/lang.c	**java_handle_option()**

- These are cc1, cc1plus, jc1 option handling routines
  - drivers just pass on arguments as declared in spec files

C Front-End

- C front-end is in gcc/ directory
  - parse entry point c_common_parse_file (c-opts.c)
    - workhorse is c_parse_file (c-parser.c)

• c-common.def	Tree codes (AST) are defined, IR codes for C compiler
• c-common.c	functions for C-like front-ends & also used for parsing C, C++ & objective C languages
• c-convert.c	contains the functions for converting C expressions to different data types. The only entry point is `convert function'.
• c-cppbuiltin.c	built-in preprocessor #defines
• c-decl.c	declaration handling
• c-dump.c	IR-dumping
• c-errors.c	pedantic warning issuance
• c-format.c	format checking for printf-like functions
• c-gimplify.c	lowering of IR (and documentation)

C Front-End …contd.

• c-incpath.c	include path generation for preprocessor
• c-lang.c	language infrastructure, front-end hookups
• c-lex.c	lexical analyzer (manually coded)
• c-objc-common.c	some functions for C and Objective-C
• c-opts.c	option processing, some init stuff
• c-parser.c	parser (based on an old bison parser)
• c-pch.c	precompiled header support
• c-ppoutput.c	preprocessing-only support (-E option)
• c-pragma.c	support for #pragma pack and #pragma weak
• c-pretty-print.c	used to pretty-print expressions in error messages
• c-semantics.c	statement list handling in IR
• c-typeck.c	functions to build IR, type checks
• gccspec.c	driver-specific tasks for gcc driver

C++ Front-End

- In subdirectory gcc/cp/
  - same parse entry point as C compiler

• call.c	function/method invocation lookup and handling
• class.c	building (the runtime artifacts of) classes *etc*.
• cp-gimplify.c	IR lowering
• cp-lang.c	language hooks for C++ front-end
• cp-objcp-common.c	common bits for C++ and Objective-C++
• cvt.c	type conversion
• cxx-pretty-print.c	C++ pretty-printer
• decl.c	declaration and variable handling
• decl2.c	additional declaration and variable handling
• dump.c	IR dumping
• rtti.c	support for run-time type information
• search.c	type search in the presence of multiple inheritance
• semantics.c	semantic checking
• tree.c	C++ front-end specific IR functionality

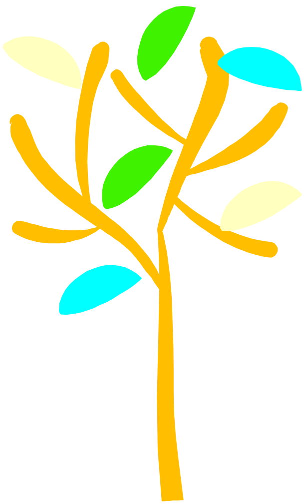
C++ Front-End ….contd.

• error.c	C++ error-reporting callbacks
• except.c	C++ exception-handling support
• expr.c	IR lowering for C++
• friend.c	C++ “friend” support
• init.c	data initializers and constructors
• lex.c	the C++ lexical analyzer
• mangle.c	C++ name mangling
• method.c	method handling; default constructor generation
• name-lookup.c	context-aware name (type, var, namespace) lookup
• optimize.c	constructor/destructor cloning
• parser.c	the C++ parser
• pt.c	parameterized type (template) support
• ptree.c	IR pretty-printing
• repo.c	C++ template repository support
• typeck.c	functionality dealing with types, conversion
• typeck2.c	types, conversion, type errors
• g++spec.c	driver-specific tasks for g++ driver

GCC Front-End ....contd.

The "treelang" front end: Essential front-end components

- configure fragment (config-lang.in)
- language-specific options (lang.opt)
- filename handling for driver (lang-specs.h)
- treelang-specific tree codes (treelang-tree.def)
- front-end hookups to toplev.c (treetree.c)
  - see gcc/langhooks.h for documentation
- flex scanner (lex.l)
- bison parser (parse.y)
- structural functions (tree1.c)



In Greater Depth

GCC New Front-End

...An Addition

GENERIC Trees

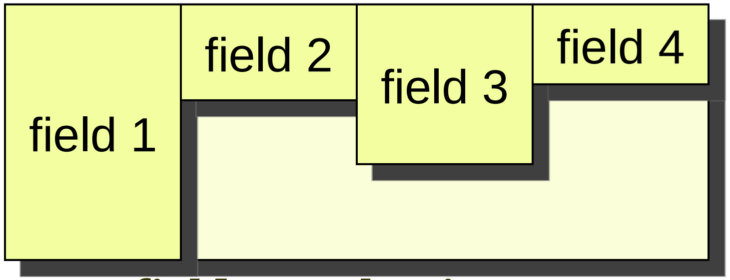
- *Front-ends are written in C !*

- We'd like to have…
  - tree node base class
    - subclasses for expressions etc.

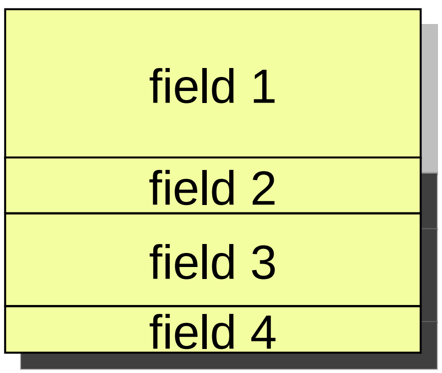
low memory

- Instead we have
  - union tree_node (gcc/tree.h)
    - each field is a struct components of union

union



struct

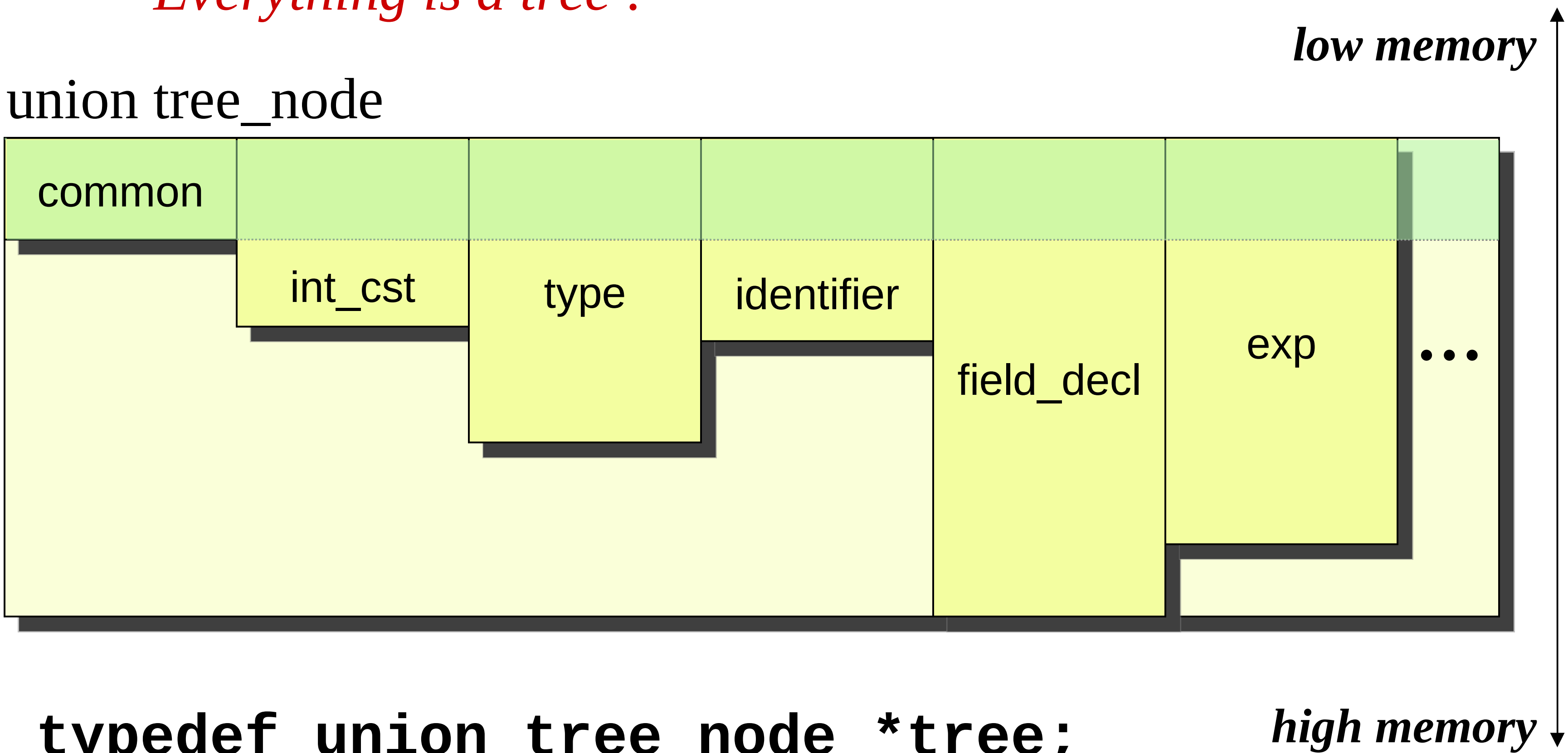


fields overlap in memory;
you're on your own for type safety !

high memory

The tree_node union

Everything is a tree !




```
typedef union tree_node *tree;
```

The tree_node union

- The "common" part contains
  - code (kind of tree – declaration, expression, etc.)
  - chain (for linking trees together)
  - type (type of the represented item – also a tree)
  - flags
    - side effects
    - addressable
    - access flags (used for other things in non-declarations)
    - 7 language-specific flags

Macros for accessing tree parts

- In the common part
  - TREE_*
    - TREE_CODE(*tree*)
    - TREE_TYPE(*tree*)
    - TREE_SIDE_EFFECTS(*tree*) *etc.*
- For specific trees
  - type trees
    - TYPE_*
      - TYPE_FIELDS(*tree*) gets a list of fields in the type
      - TYPE_NAME(*tree*) gets the type's associated decl

Expression trees

- Lots of tree codes used for expressions
  - gcc/tree.def defines all standard tree codes
  - LT_EXPR less-than conditional
  - TRUTH_ORIF_EXPR short-circuiting OR conditional
  - MODIFY_EXPR assignment
  - NOP_EXPR type promotion (typically)
  - SAVE_EXPR store in temporary for multiple uses
  - ADDR_EXPR take address of
- Front-end extensions to GENERIC permitted
  - gcc/c-common.def
  - gcc/cp/cp-tree.def *e.g.* DYNAMIC_CAST_EXPR
  - gcc/java/java-tree.def *e.g.* SYNCHRONIZED_EXPR

A few useful front-end functions

- **build()** expression tree building – pass tree code, tree type, and (arbitrary number of) operands
- **fold()** simple tree restructuring and optimization; mostly useful for constant folding
- **gcc_assert()** assertion verification – if it fails it gives an "internal compiler error" report with source file and line number under compilation (as well as source file and line number in compiler code)

Code Naming Conventions

- Preprocessor macros ALL UPPERCASE
- Variables/functions all lowercase with underscores
- Predicates end in "_P" or "_p"
- Global flags start with "flag_"
- Global trees (vary somewhat with front-end)
  - null_node (or null_pointer_node)
  - integer_zero_node
  - void_type_node
  - integer_unsigned_type_node (or unsigned_int_type_node)
- Tree accessor macros *FROM_TO* (*e.g.* TYPE_DECL)

GCC Front-End ....Generic & Gimple

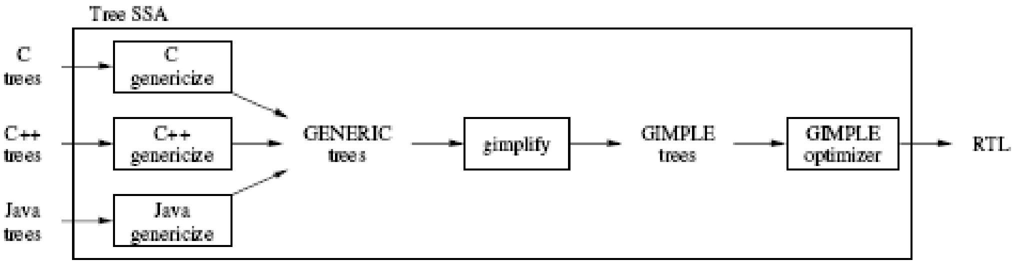
- GENERIC is a common representation shared by all front ends
  - Parsers may build their own representation for convenience
  - Once parsing is complete, they emit GENERIC

- Gimple is a simplified version of GENERIC
  - 3-address representation
  - Restricted grammars to facilitate the job of optimizers

GENERIC	High GIMPLE	Low GIMPLE
if (foo (a + b,c))	t1 = a + b	t1 = a + b
c = b++ / a	t2 = foo (t1, c)	t2 = foo (t1, c)
endif	if (t2 != 0)	if (t2 != 0) <**L1,L2**>
return c	t3 = b	**L1:**
	b = b + 1	t3 = b
	c = t3 / a	b = b + 1
	endif	c = t3 / a
	return c	goto **L3**
		L2:
		L3:

GCC Front-End ....GIMPLE

- No hidden/implicit side-affects
- Simplified Control Flow
  - Loops represented with if/goto
  - Lexical scopes removed (low-Gimple)



- Locals of scalar types are treated as “registers” (real operands)

- Globals, aliased variables & non-scalar types treated as “memory”(virtual operands)

- At most one memory load/store operation per statement
  - Memory loads only on RHS of assignments
  - Stores only on LHS of assignments

- Can be incrementally lowered (2 levels currently)
  - High GIMPLE -> lexical scopes & inline parallel regions
  - Low GIMPLE -> no-scopes & out-of-line parallel regions

- It contains extensions to represent explicit parallelisms (OpenMP)

GCC Front-End ....GIMPLE Statements

- GIMPLE statements are instances of type tree
- Every block contains a double-linked list of statements
- Manipulations done through iterators

```
block_statement_iterator si;
basic_block bb;
FOR_EACH_BB(bb)
  for (si = bsi_start(bb); !bsi_end_p(si); bsi_next(&si))
   print_generic_stmt (stderr, bsi_stmt(si), 0);
```

- Real operands (DEF, USE)
  - Non-aliased, scalar, local variables
  - Atomic references to the whole object
  - GIMPLE “registers” (may not fit in a physical register)
- Virtual or memory operands (VDEF, VUSE)
  - Globals, aliased, structures, arrays, pointer dereferences
  - Potential and/or partial references to the object
  - Distinction becomes important when building SSA form

GCC Front-End ....GIMPLE Statement Operands

- Real operands are part of the statement

```
int a, b, c
c = a + b
```

- Virtual operands are represented by two operators `VDEF` and `VUSE`

```
int c[100]
int *p = (i > 10) ? &a : &b
# a = VDEF <a>
# b = VDEF <b>
# VUSE <c>
*p = c[i]
```

a or b <u>may</u> be defined

c[i] is a <u>partial</u> load from c

```
use_operand_p use;
ssa_op_iter i;
FOR_EACH_SSA_USE_OPERAND (use, stmt, i, SSA_OP_ALL_USES)
   {
       tree op = USE_FROM_PTR (use);
       print_generic_expr (stderr, op, 0);
   }
```

- Prints all `USE` and `VUSE` operands from `stmt`
- `SSA_OP_ALL_USES` filters which operands are of interest during iteration
- For `DEF` and `VDEF` operands, replace “use” with “def” above

GCC – Control Flow Graph (CFG)

- A **control flow graph (CFG)** is an abstract data structure used in compilers. It is an abstract representation of a procedure or program, maintained internally by a compiler.

- Each node in the graph represents a basic block. There are two specially designated blocks: the *entry block*, through which control enters into the flow graph, and the *exit block*, through which all control flow leaves.

- CFG example – Consider the following fragment of code:

  0: (A) t0 = read_num

  1: (A) if t0 mod 2 == 0 goto 4

  2: (B) print t0 + " is odd."

  3: (B) goto 5

  4: (C) print t0 + " is even."

  5: (D) end program

GCC Front-End ....CONTROL FLOW GRAPH (CFG)

- Built early during lowering & survives until late in RTL (right before machine dependent transformations *pass_machine_reorg*)
- In GIMPLE, instruction stream is physically split into blocks
  - All jumps instructions replaced with edges
- In RTL, CFG is laid out over double-linked instruction stream
  - Jumps instructions preserved
- Every CFG accessor requires a struct function argument
- In intraprocedural mode, accessors have shorthand aliases that use cfun by default
- CFG is an array of double-linked blocks
- Same data structures used for GIMPLE & RTL

Demystifying GCC

Front-end **Middle-end** Back-end

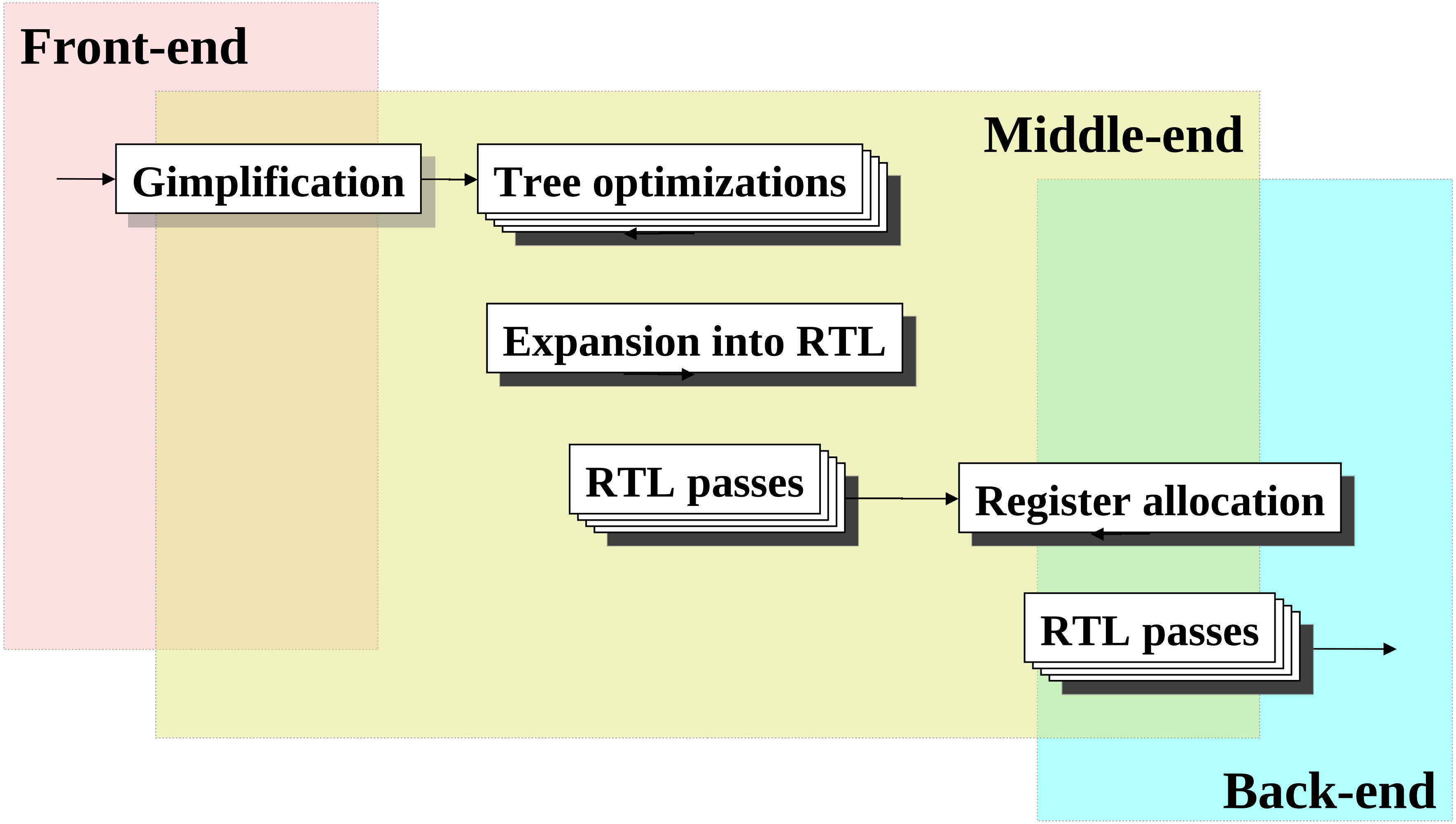
GCC Middle-End

Part III

Optimization of trees

Static Single-Assignment Intermediate Representation

Middle-End Context



Optimizations over the Tree representation

- Managed by *pass manager* in gcc/passes.c
  - init_optimization_passes orders the passes
  - passes represented by a **tree_opt_pass** struct (tree-pass.h) even though it does RTL now too
    - "gate" function – whether or not to run optimization
    - "execute" function – implementation of pass
    - property bitmaps
      - properties required, destroyed, and created
    - "todo" bitmaps
      - run internal GC, dump the tree, verify SSA form, *etc.*

Debugging Middle-End tree passes

- **Command-line options for dumping trees:**
  - **-fdump-tree-*X*** — **output after pass *X***
  - **-fdump-tree-original** — **output initial tree (before all opts)**
  - **-fdump-tree-optimized** — **output final GIMPLE (after all opts)**
  - **-fdump-tree-gimple** — **dump before & after gimplification**
  - **-fdump-tree-inlined** — **output after function inlining**
  - **-fdump-tree-all** — **output after each pass**
- **(Make sure you specify an –O level or you might not get anything.)**

- **Passes available for dumping in GCC 4.1.1 (see info page):**

cfg, vcg, ch, ssa, salias, alias, ccp, storeccp, pre, fre, copyprop, store_copyprop, dce, mudflap, sra, sink, dom, dse, phiopt, forwprop, copyrename, nrv, vect, vrp

Debugging Middle-End tree passes

•Can specify options for tree dumps:

• address	print address of each tree node
• slim	less output; don't dump all scope bodies
• raw	raw tree output (rather than pretty-printed C-like trees)
• details	detailed output (not supported by all passes)
• stats	statistics (not supported by all passes)
• blocks	basic block boundaries
• vops	output virtual operands for each statement
• lineno	output line #s
• uid	output decl's unique ID along with each variable
• all	all except raw, slim, and lineno\

e.g.

-fdump-tree-dse-details	**detailed post-DSE output**
-fdump-tree-all-all	**(almost) everything**

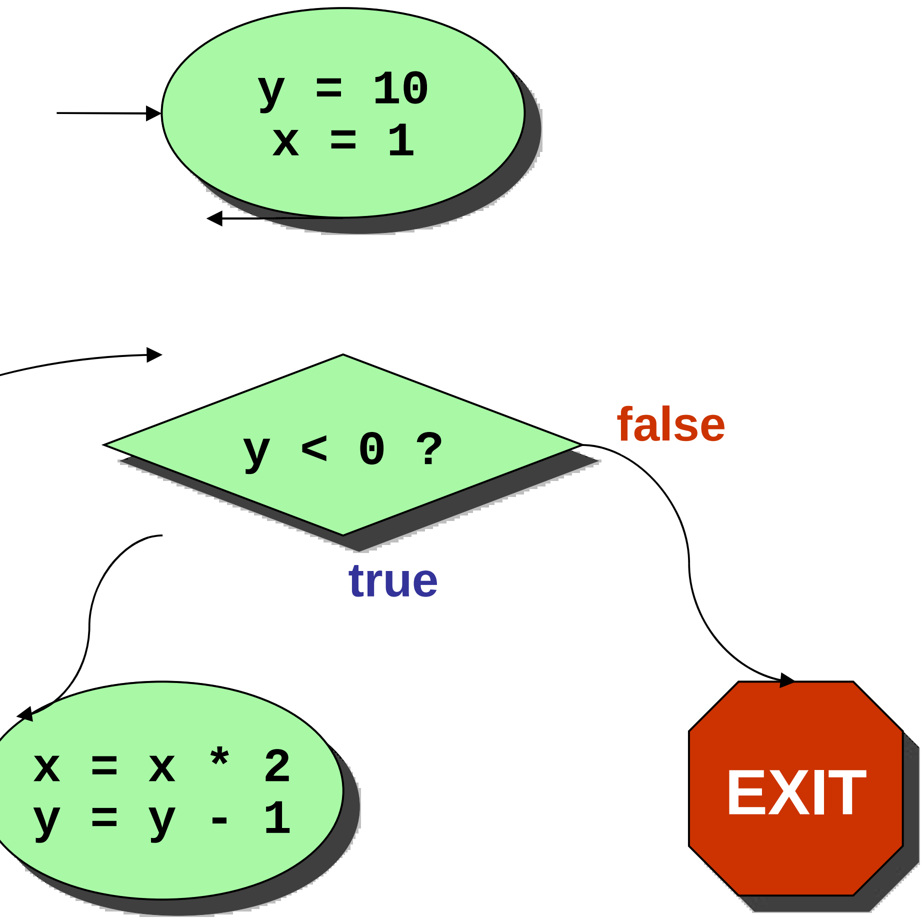
Static Single-Assignment (SSA) form

Cytron *et al*. *Efficiently computing static single assignment form and the control dependence graph*. ACM *TOPLAS*, October 1991.

- (Pure) functional languages have nice properties for optimization
- *single-assignment:* one assignment to each variable
- *static single-assignment:* next best thing
    - each variable assigned at one static location in the program
- makes it clearer where data is produced
    - reduces complexity of many optimization algorithms
    - removes association of variable uses over its lifetime

SSA renaming (1)

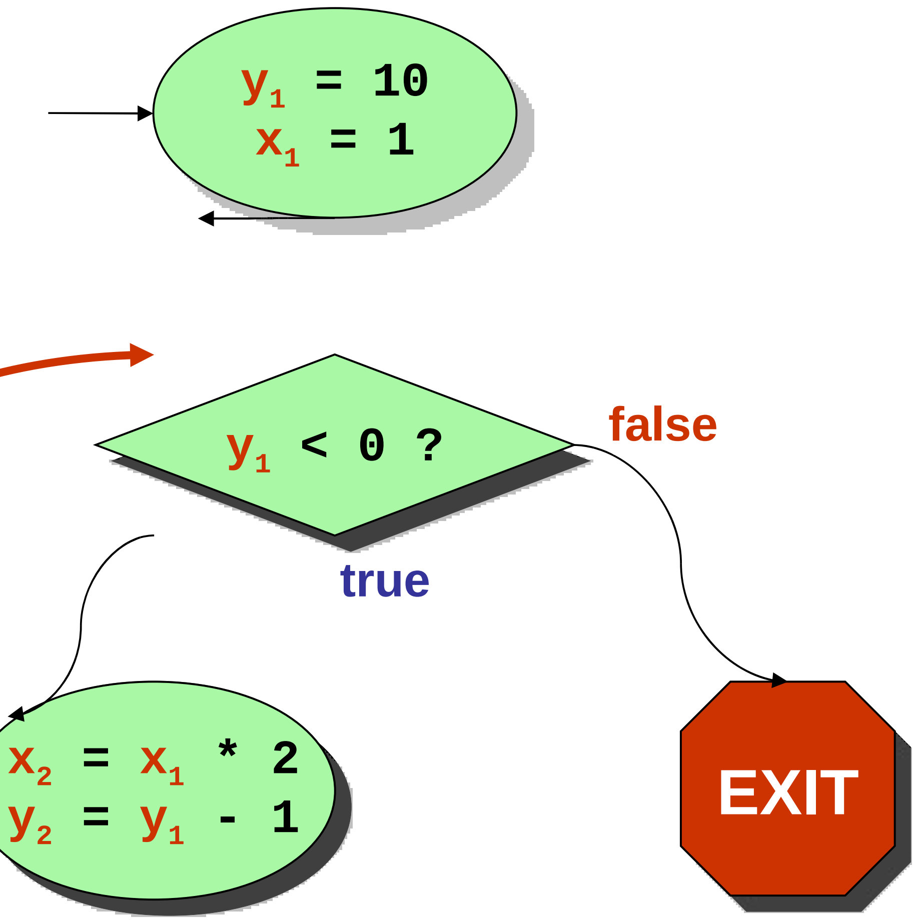
```
y = 10;

/* compute 2^y */
x = 1;
while (y > 0) {
  x = x * 2;
  y = y - 1;
}
```



model control flow

SSA renaming (2)

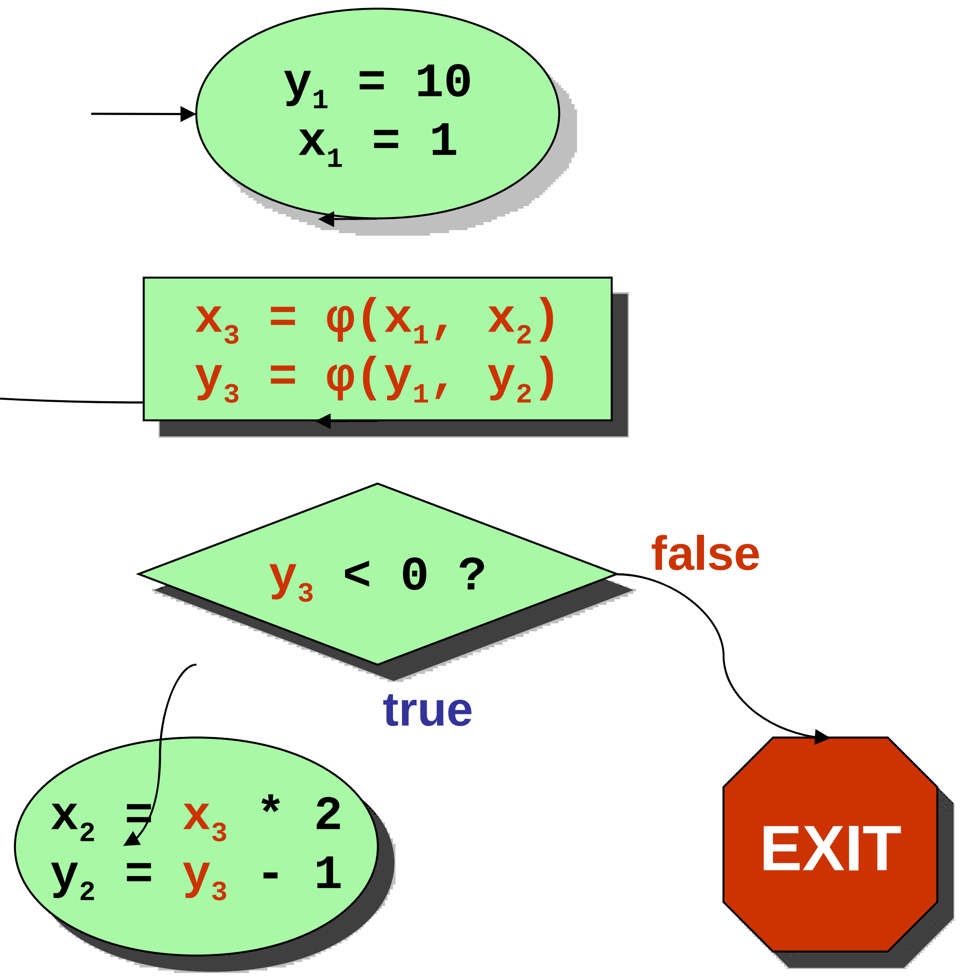
```
y1 = 10;

/* compute 2^y */
x1 = 1;
while (y1 > 0) {
   x2 = x1 * 2;
   y2 = y1 - 1;
}
```



version all variables

SSA renaming (3)

```
y1 = 10;

/* compute 2^y */
x1 = 1;
while(true) {
   x3 = φ(x1, x2);
   y3 = φ(y1, y2);

   if (y3 > 0)
     break;

   x2 = x3 * 2;
   y2 = y3 - 1;
}
```

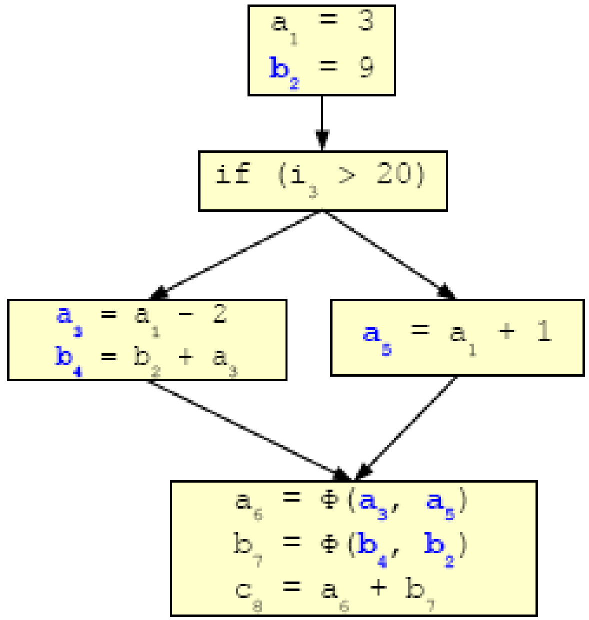


insert "phi" nodes

GCC Middle-End SSA Form

Static Single Assignment (SSA)

- Versioning representation to expose data flow explicitly
- Assignments generate new versions of symbols
- Convergence of multiple versions generates new one (Φ functions)



- Rewriting (or standard) SSA form
  - Used for real operands
  - Different names for the same symbol are *distinct objects*
  - overlapping live ranges (OLR) are allowed

```
if (x2 > 4)
  z5 = x3 - 1
```

  - Program is taken out of SSA form for RTL generation (new symbols are created to fix OLR)

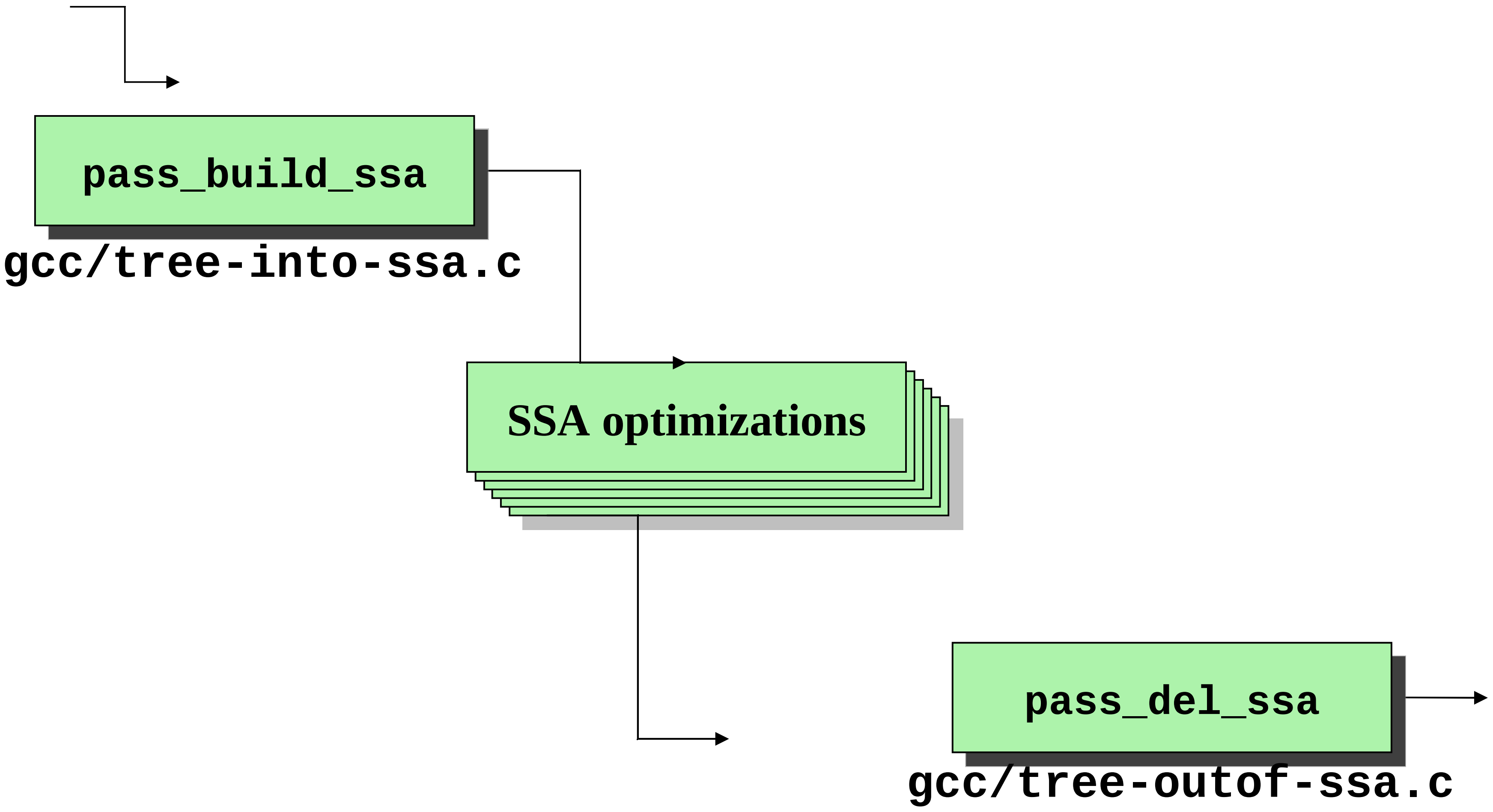
GCC Middle-End SSA Form ....contd.

- Factored Use-Def Chains (FUD Chains)
  - Also known as Virtual SSA Form
  - Used for virtual operands
  - All names refer to the *same object*
  - Optimizers may not produce OLR for virtual operands
- Both SSA forms can be updated incrementally
  - Name→name mappings
  - Individual symbols marked for renaming

- `VDEF` operand needed to maintain DEF-DEF links
- They also prevent code movement that would cross stores after loads
- When alias sets grow too big, static grouping heuristic reduces number of virtual operators in aliased references

```
foo (i, a, b, *p)
{
  p_2 = (i_1 > 10) ? &a : &b

  # a_4 = VDEF <a_11>
  a = 9;

  # a_5 = VDEF <a_4>
  # b_7 = VDEF <b_6>
  *p_2 = 3;

  # VUSE <a_5>
  t1_8 = a;

  t3_10 = t1_8 + 5;
  return t3_10;
}
```

Into and out of SSA form in GCC



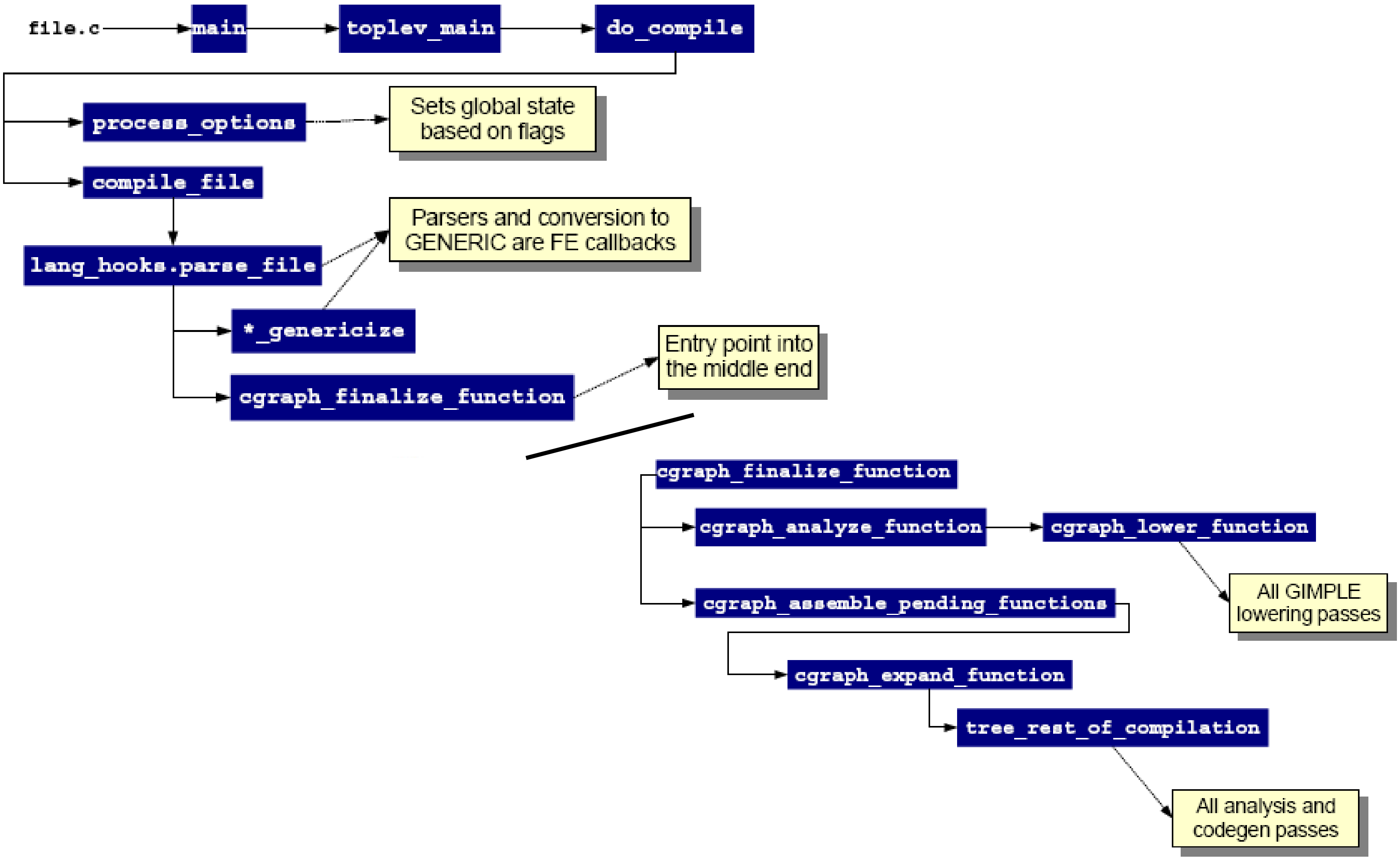
Dealing with SSA form in GCC

- Given a tree node *n* with code = PHI_NODE
  - PHI_RESULT(*n*) — *get lhs of φ*
  - PHI_NUM_ARGS(*n*) — *get rhs count*
  - PHI_ARG_DEF(*n*, *i*) — *get ssa-name*
  - PHI_ARG_EDGE(*n*, *i*) — *get edge*
  - PHI_ARG_ELT(*n*, *i*) — *tuple (ssa-name, edge)*
- Given a tree node *n* with code = SSA_NAME
  - SSA_NAME_DEF_STMT(*n*) — *get defining statement*
  - SSA_NAME_VERSION(*n*) — *get SSA version #*

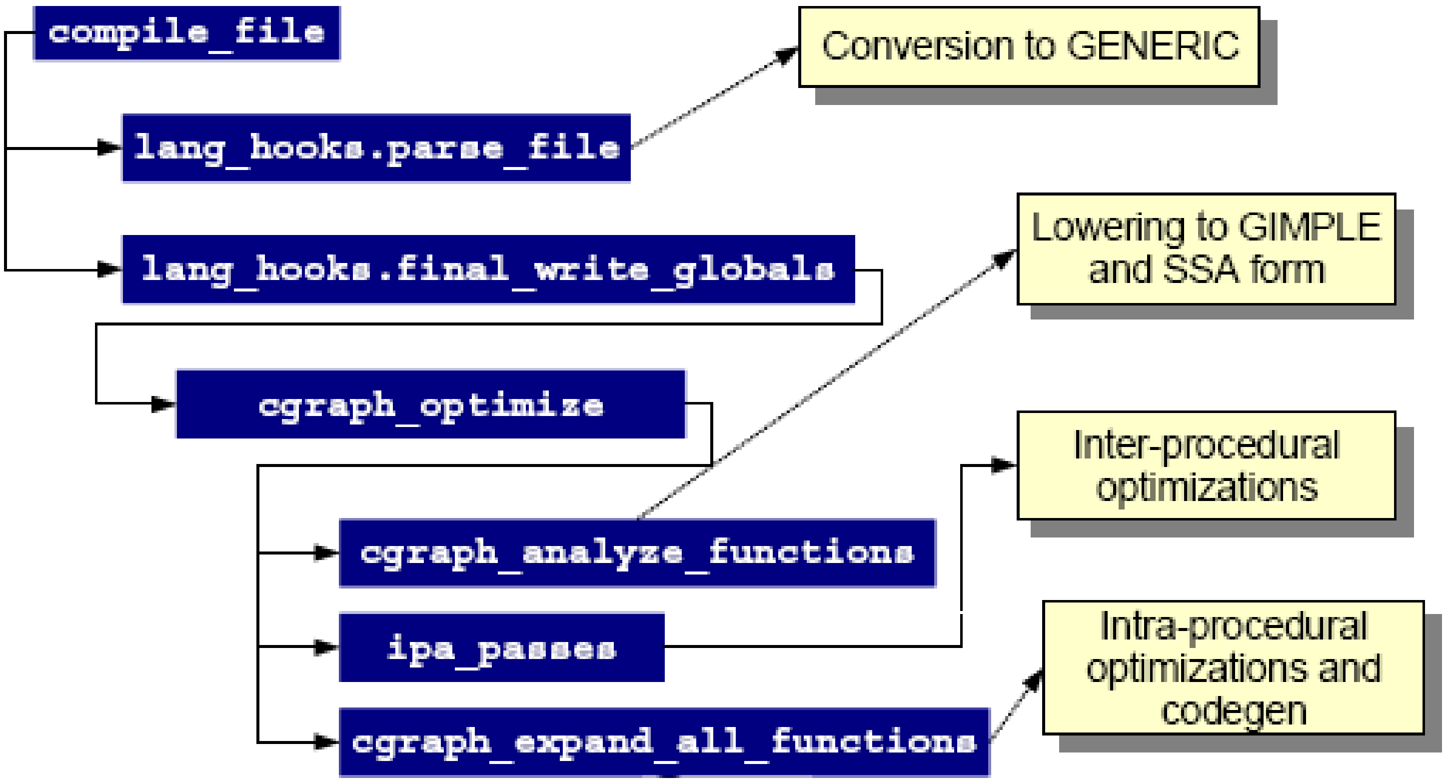
GCC Middle-End SSA Optimizers

- Operate on GIMPLE
- Around 100 passes
  - Vectorization
  - Various loop optimizations
  - Traditional scalar optimizations: CCP, DCE, DSE, FRE, PRE, VRP, SRA, jump threading, forward propagation
  - Field-sensitive, points-to alias analysis
  - Pointer checking instrumentation for C/C++
  - Interprocedural analysis and optimizations: CCP, inlining, points-to analysis, pure/const and type escape analysis

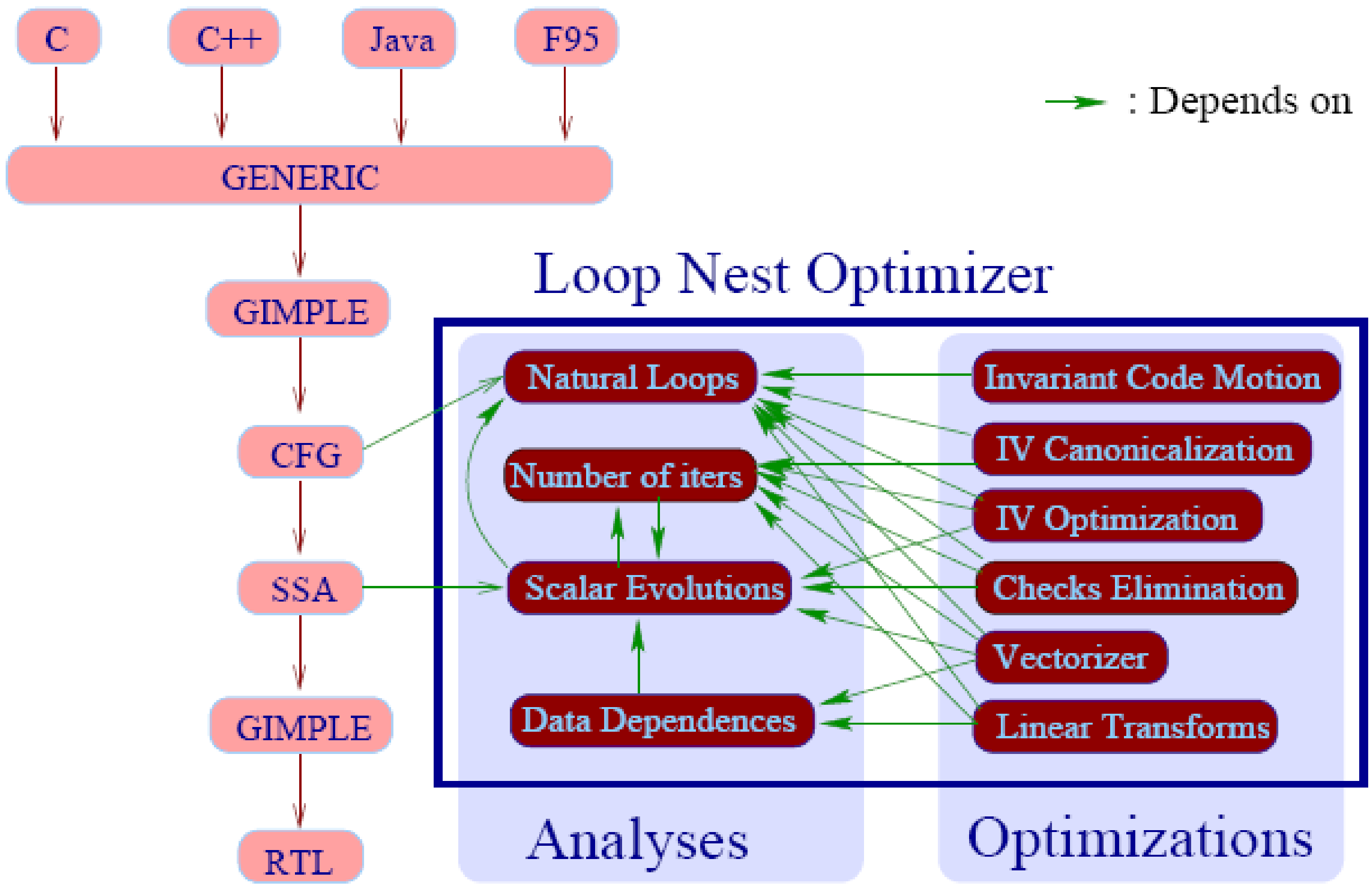
Optimizations .... Compilation Flow (O0)



Optimizations .... Compilation Flow (O1+)



Loop Transformations



Demystifying GCC

Front-end Middle-end **Back-end**

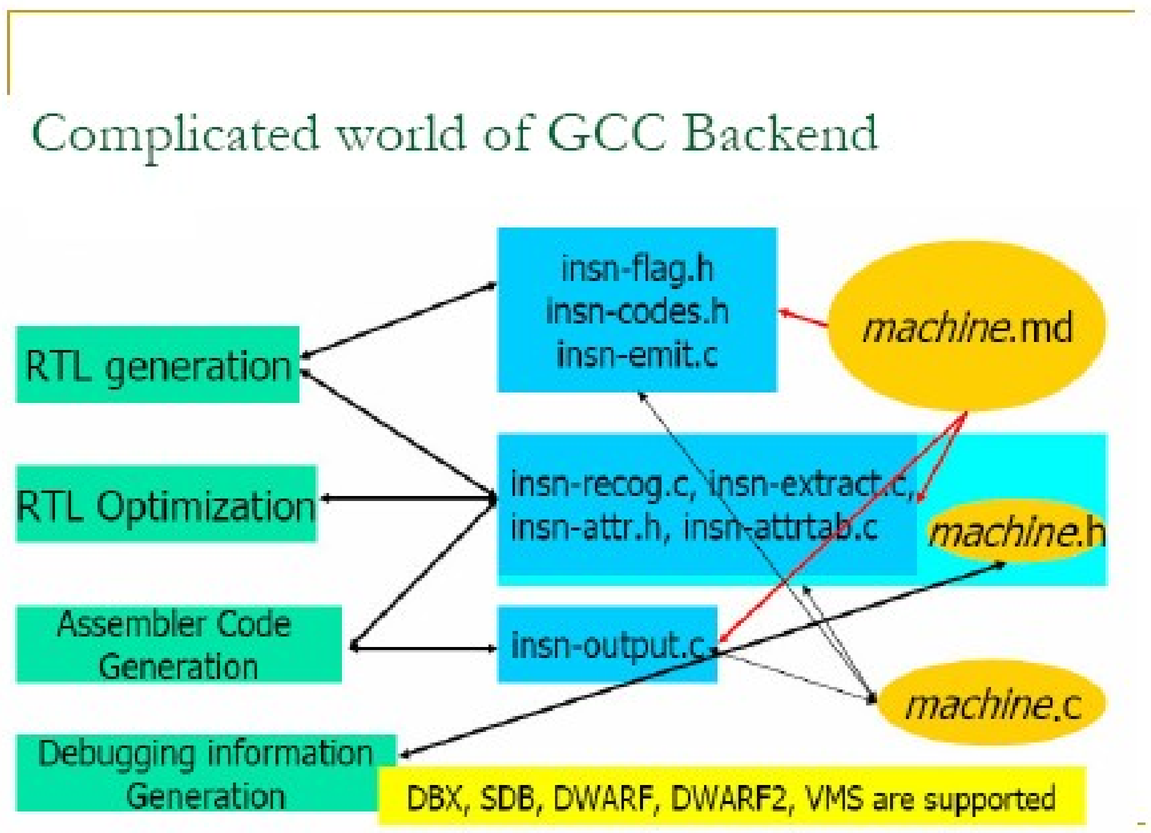
GCC Back-End

Part IV

Machine – Descriptions
Code Generation

RTL Machine-Descriptor RISC-vs.-CISC

GCC Backend


Complicated world of GCC Backend
RTL generation
RTL Optimization
Assembler Code Generation
Debugging information Generation
insn-flag.h
insn-codes.h
insn-emit.c
insn-recog.c, insn-extract.c, insn-attr.h, insn-attrtab.c
insn-output.c
machine.md
machine.h
machine.c
DBX, SDB, DWARF, DWARF2, VMS are supported

Source-Code Organization

- GCC Source Tree – Target independent files are in gcc/*.{c, h, def}

File	Purpose
gcc/tree.def	Definition of known tree-level idioms
gcc/rtl.def	Definition of known RTL operators
gcc/optabs.h	Declaration of operator tables for tree-to-RTL translations
gcc/optabs.c	Definition of operator tables & intrinsic, basic support functions
gcc/fold-const.c	Tree folding routines
gcc/expr.c	Expansion of Trees into RTL expressions

- Target specific support – It’s in “gcc/config/[target_family]”
  - sufficient powerful to support a new target
  - size: variable, few KLOC to few tens of KLOC

Target-Specific Files

- Machine-description: <target>.md
  - Definition of RTL instructions and their translation to assembly
- Target-specific compiler options: <target>.opt
  - Command-line options of GCC specific to the target
- Target-specific definitions: <target>.h
  - Basic parameters and features
- Target-specific support functions: <target>.c
  - Target predicates, code-generation functions, target variants
- Makefile fragments: t-<target>
  - Features of build, e.g. multiple versions of libgcc

Machine-Descriptor

Machine Descriptor – *<processor>.md ..1*

- CPU description
- Functional Units, Latency and etc
- RTL Patterns
- Used when convert Tree into RTL
- All kind of RTL Patterns which can be generated
- Assembler mnemonic
- etc.

Machine-Descriptor ....contd.

Machine Descriptor – *<processor>.md ..2*

- Describe insns names for the generate pass, using define_insn (or may also use define_expand) and has 5 major parts
  - Name
  - RTL Template
  - Conditions
  - Output
  - Attributes

```
(define_insn "movsi"
  [(set (match_operand:SI 0 "nonimmediate_operand" "=r,r,m,r,r,r")
          (match_operand:SI 1 "general_operand"      "r,m,r,I,K,i"))]
  ""
  "*
  switch(which_alternative)
   {
    case 0:     return \"l.ori   \\t%0,%1,0\\t # move reg to reg\";
    case 1:     return \"l.lwz    \\t%0,%1\\t # SI load\";
    case 2:     return \"l.sw    \\t%0,%1\";
    case 3:     return \"l.addi  \\t%0,r0,%1\\t # move immediate\";
    case 4:     return \"l.ori   \\t%0,r0,%1\\t # move immediate\";
    case 5:     return \"l.movhi \\t%0,hi(%1)\\t # move immediate (high);
                       l.ori  \\t%0,%0,lo(%1)\\t # move immediate (low)\";
    default:    return \"invalid alternative\";
   }
  "
  [(set_attr "type" "add,load,store,add,logic,move")
   (set_attr "length" "1,1,1,1,1,2")])
```

Machine-Descriptor ....contd.

Machine Description file & RTL

```
(define_insn "movsi"
 [(set (match_operand:SI 0 "nonimmediate_operand" "=r,r,m,r,r,r")
                (match_operand:SI 1 "general_operand"      "r,m,r,I,K,i"))]
 ""
 "*
 switch(which_alternative)
  {
   case 0:      return \"l.ori   \\t%0,%1,0\\t # move reg to reg\";
   case 1:      return \"l.lwz   \\t%0,%1\\t # SI load\";
   case 2:      return \"l.sw    %1%0,%1\";
   case 3:      return \"l.addi  \\t%0,r0,%1\\t # move immediate\";
   case 4:      return \"l.ori   \\t%0,r0,%1\\t # move immediate\";
   case 5:      return \"l.movhi \\t%0,hi(%1)\\n\\t # move immediate
                        l.ori \\t%0,%0,lo(%1)\\n\\t # move imm
   default:    return \"invalid alternative\";
  }
 [(set_attr "type" "add,load,store,add,logic,move")
  (set_attr "length" "1,1,1,1,1,2")])
```

The Parse tree has defined pattern names which are converted into RTL insn list based on named instruction patterns (define_inst) e.g. "movsi"

- Name : movsi means "move op2 to op1" in SI mode
- RTL Template: Match the type of Op1 & 2
- Output: Based on the match one of the code is substituted
- Attribute : Attribute of the instruction which can follow the current instruction.

RTL – Register Transfer Language

- RTL ≈ assembler for an abstract machine with infinite registers

- Represents low level features – Register Classes
  - Memory addressing modes
  - Word sizes & types
  - Compare-&-branch Instructions
  - Calling conventions
  - Bitfield operations
  - Type & sign conversions

- Commonly represented in LISP-like form

- Operands do not have types, but types modes (SI Modes, 4-byte integers)

```
b = a - 1
```

```
(set (reg/v:SI 59 [ b ])
     (plus:SI (reg/v:SI 60 [ a ]
              (const int -1 [0xffffffff]))))
```

RTL Optimizer

- Around 70 passes
- Operate closer to the target
  - Register allocation
  - Scheduling
  - Software pipelining
  - Common subexpression elimination
  - Instruction recombination
  - Mode switching reduction
  - Peephole optimizations
  - Machine specific reorganization

- When RTL patterns are not enough?

RTL Statements

- RTL statements (insns) are instances of type `rtx`
- Unlike GIMPLE statements, RTL insns contain embedded links
- Six types of RTL insns

`INSN`	Regular, non-jumping instruction
`JUMP_INSN`	Conditional and unconditional jumps
`CALL_INSN`	Function calls
`CODE_LABEL`	Target label for `JUMP_INSN`
`BARRIER`	Control flow stops here
`NOTE`	Debugging information

- Some elements of an RTL insn

`PREV_INSN`	Previous statement
`NEXT_INSN`	Next statement
`PATTERN`	Body of the statement
`INSN_CODE`	Number for the matching machine description pattern (-1 if not yet recog'd)
`LOG_LINKS`	Links dependent insns in the same block Used for instruction combination
`REG_NOTES`	Annotations regarding register usage

RTL Statements .... Contd.

- Traversing all RTL statements

```
basic_block bb;
FOR_EACH_BB (bb)
   {
      rtx insn = BB_HEAD (bb);
      while (insn != BB_END (bb))
         {
            print_rtl_single (stderr, insn);
            insn = NEXT_INSN (insn);
         }
   }
```

- No operand iterators, but RTL expressions are very regular
- Number of operands and their types are defined in `rtl.def`

`GET_RTX_LENGTH`	Number of operands
`GET_RTX_FORMAT`	Format string describing operand types
`XEXP/XINT/XSTR/...`	Operand accessors
`GET_RTX_CLASS`	Similar expressions are categorized in classes

RTL Statements

- Operands and expressions have modes, not types
- Supported modes will depend on target capabilities
- Some common modes

`QImode`	Quarter Integer (single byte)
`HImode`	Half Integer (two bytes)
`SImode`	Single Integer (four bytes)
`DImode`	Double Integer (eight bytes)
...	

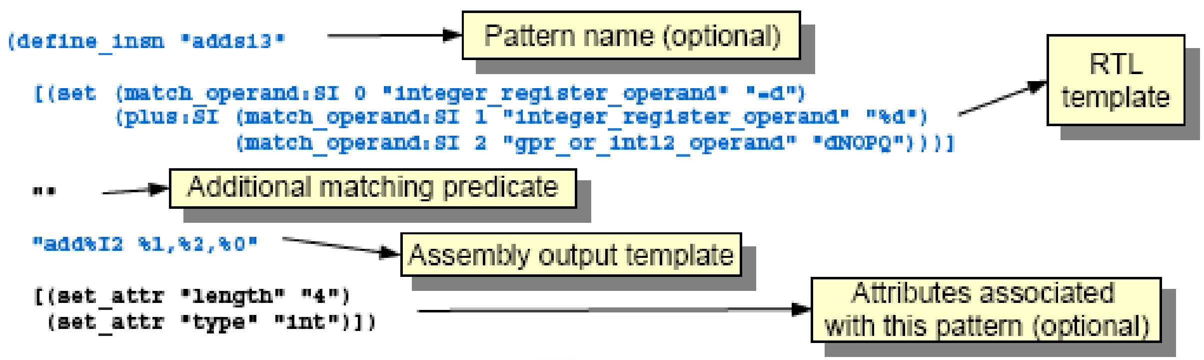
- Modes are defined in `machmode.def`

Code-Generation

- Code is generated using a rewriting system
- Target specific configuration files in

  `gcc/config/<arch>`

- Three main target-specific files
  - `<arch>.md` Code generation patterns for RTL insns
  - `<arch>.h` Definition of target capabilities (register classes, calling conventions, type sizes, etc)
  - `<arch>.c` Support functions for code generation, predicates and target variants
  - Two main types of rewriting schemes supported
    - Simple mappings from RTL to assembly (`define_insn`)
    - Complex mappings from RTL to RTL (`define_expand`)
  - `define_insn` patterns have five elements



Code-Generation ....contd.

`define_insn "addsi3"`

- Named patterns
  - Used to generate RTL
  - Some standard names are used by code generator
  - Some missing standard names are replaced with library calls (e.g., `divsi3` for targets with no division operation)
  - Some pattern names are mandatory (e.g. move operations)
- Unnamed (anonymous) patterns do not generate RTL, but can be used in insn combination

Code-Generation .... contd.

```
[(set (match_operand:SI 0 "integer_register_operand" "=d,=d")
      (plus:SI (match_operand:SI 1 "integer_register_operand" "%d,m")
               (match_operand:SI 2 "gpr_or_int12_operand" "dNOPQ,m")))]
```

Matching uses
- Machine mode (SI, DI, HI, SF, etc)
- Predicate (a C function)
- Both operands and operators can be matched

Constraints provide second level of matching
Select best operand among the set of allowed operands
Letters describe kinds of operands
Multiple alternatives separated by commas

`"add%I2 %1,%2,%0"`

- Code is generated by emitting strings of target assembly
- Operands in the insn pattern are replaced in the `%n` placeholders
- If constraints list multiple alternatives, multiple output strings must be used
- Output may be a simple string or a C function that builds the output string

Pattern Expansion

- Some standard patterns cannot be used to produce final target code. Two ways to handle it
  - Do nothing. Some patterns can be expanded to libcalls
  - Use `define_expand` to generate matchable RTL
- Four elements
  - The name of a standard insn
  - Vector of RTL expressions to generate for this insn
  - A C expression acting as predicate to express availability of this instruction
  - A C expression used to generate operands or more RTL

```
(define_expand "ashlsi3"
  [(set (match_operand:SI 0 "register_operand" "")
        (ashift:SI
           (match_operand:SI 1 "register_operand" "")
           (match_operand:SI 2 "nonmemory_operand" "")))]
  ""
  "{
     if (GET_CODE (operands[2]) != CONST_INT
         || (unsigned) INTVAL (operands[2]) > 3)
       FAIL;
   }")
```

  - Generate a left shift only when the shift count is [0...3]
  - `FAIL` indicates that expansion did not succeed and a different expansion should be tried (e.g., a library call)
  - `DONE` is used to prevent emitting the RTL pattern. C fragment responsible for emitting all insns.

GCC ....IPA

- Intra-Procedural Analysis (IPA) – IPA & optimizations is about optimizing across function boundaries. Basic optimizations includes –
  - Removal of unused functions & variables
  - Alias analysis
  - De-virtualization
  - Inlining
  - Constant propagation
  - Register Allocation
  - Memory consumption problems
  - Compile-time problems, etc

RISC & CISC

- Reduced Instruction Set Computer
- Key features of RISC
  - Large number of **general purpose registers**
    - And/or use of compiler technology to optimize register use
  - Limited and **simple instruction set**
  - Emphasis on **optimizing the instruction pipeline**
- CISC (Complex Instruction Set Computer)
  - Each instruction executes several low-level operations (load from memory, arithmetic, store,…)
  - Coined after RISC came out

Intention of CISC

- Ease compiler writing
  - Hardware implementations of **HLL statements**
    - e.g. CASE (switch) on VAX, loop
  - **Large instruction sets**
  - **More addressing modes**
- Improve execution efficiency
  - Complex operations can be implemented in microcode

- *Well, did it work as intended?*

Comparison of processors

	Complex Instruction Set (CISC)Computer			Reduced Instruction Set (RISC) Computer		Superscalar		
Characteristic	IBM 370/168	VAX 11/780	Intel 80486	SPARC	MIPS R4000	PowerPC	Ultra SPARC	MIPS R10000
Year developed	1973	1978	1989	1987	1991	1993	1996	1996
Number of instructions	208	303	235	69	94	225		
Instruction size (bytes)	2–6	2–57	1–11	4	4	4	4	4
Addressing modes	4	22	11	1	1	2	1	1
Number of general-purpose registers	16	16	8	40 - 520	32	32	40 - 520	32
Control memory size (Kbits)	420	480	246	—	—	—	—	—
Cache size (KBytes)	64	64	8	32	128	16-32	32	64

Did CISC Work?

1. **Simpler Compilation?**
   - Complex machine instructions **harder to exploit**
     - E.g., orthogonal addressing modes were ignored commonly
   - Optimization more difficult

2. **Smaller programs?**
   - Program takes up less memory but…
     - Memory is now cheap. No point for saving.
   - May ***not*** occupy **less bits**, it just look **shorter in symbolic form**

Did CISC Work?

3. **Faster programs?**
   - **Bias** towards use of **simpler instructions**
     - Advantage of complex instruction (multiple instructions bundled in one complex instruction) is not realized
     - E.g., orthogonal addressing modes were ignored commonly
   - **More complex control unit** & larger microprogram control store
     - thus simple instructions take **longer to execute**

- It is far from clear that CISC is the appropriate solution

RISC Rationale

- Since real-world programs execute very simple operations, **make** those **common operations** as simple and **as fast as possible**.
- Make instructions simple, so **each one** could be executed in a **single clock cycle**
  - Instead of single complex instruction, use multiple simpler instructions
- The clock rate of the CPU is limited by the *slowest* instruction, so **speed up** that instruction
  - How? Maybe by **reducing the number of addressing modes** it supports
  - Clock cycle is reduced and **every instruction get faster**

RISC Characteristics

1. One instruction per cycle
2. Register to register operations
3. Few, simple addressing modes
4. Few, simple instruction formats

RISC vs. CISC

- RISC problems
  - **Total number of instructions** read from the memory is **larger**
- RISC vs. CISC
  - Not clear cut
- Many designs borrow from both philosophies
  - e.g. PowerPC and Pentium II (and later)
    → no more *pure* RISC or CISC

RISC vs. CISC

		Decode Complexity			Ease of pipelining		Compiler	
	Processor	Number of instruction sizes	Max instruction size in bytes	Number of addressing modes	Max number of memory operands	Max Number of MMU uses	Number of bits for integer register specifier	Number of bits for FP register specifier
RISC	AMD29000	1	4	1	1	1	8	3[a]
	MIPS R2000	1	4	1	1	1	5	4
	SPARC	1	4	2	1	1	5	4
	MC88000	1	4	3	1	1	5	4
	HP PA	1	4	10[a]	1	1	5	4
	IBM RT/PC	2[a]	4	1	1	1	4[a]	3[a]
	IBM RS/6000	1	4	4	1	1	5	5
	Intel i860	1	4	4	1	1	5	4
CISC	IBM 3090	4	8	2[b]	2	4	4	2
	Intel 80486	12	12	15	2	4	3	3
	NSC 32016	21	21	23	2	4	3	3
	MC68040	11	22	44	2	8	4	3
	VAX	56	56	22	6	24	4	0
RISC + CISC	Clipper	4[a]	8[a]	9[a]	1	2	4[a]	3[a]
	Intel 80960	2[a]	8[a]	9[a]	1	—	5	3[a]

What to do if you find a bug in GCC

- Check to see if bug is present in SVN version
- Check to see if bug is in bug database
  - http://gcc.gnu.org/bugzilla/
- Collect version information (gcc --version)
- Guidelines: http://gcc.gnu.org/bugs.html
- Report it: http://gcc.gnu.org/bugzilla

Queries

Queries ??

Mukkaysh Srivastav
srimks@msn.com