CHEMNITZ UNIVERSITY
OF TECHNOLOGY

Faculty of Computer Science

Chair for Real Time Systems

Diploma Thesis

Porting the GCC-Backend
to a VLIW-Architecture

Author:	Jan Parthey
Supervisor:	Dr. Robert Baumgartl
Date of Submission:	01.03.2004

Jan Parthey

Porting the GCC-Backend to a VLIW-Architecture

Diploma Thesis, Chemnitz University of Technology, 2004

Abstract

Digital Signal Processors (DSP) are widely used today, e.g. in sound and video processing. In order to accelerate the design process of applications, large parts of DSP programs can normally be written in C and only a few performance-critical functions need to be written in Assembler. Of course, an essential prerequisite for this way of software design is a C compiler with target support for the envisaged DSP platform.

The TMS320-C6000, henceforth denoted as ‘C6x’, is a DSP series designed by Texas Instruments. Commercial C compilers for the ‘C6x’ are available from the manufacturer. However, as explained below, there would be many advantages in having a compiler available that would be free in the sense defined by the Free Software Foundation [3]. At the time of starting this work, the author was (and still is) not aware of any such compiler being available and therefore deemed it worthwhile to make an effort in this direction.

This diploma thesis discusses the implementation of basic support for the C6x platform into GCC 3.3 [11], which is achieved by using the mechanisms provided by GCC for porting to new target platforms. As a prerequisite, fundamental concepts of the inner workings of GCC have to be investigated. The documentation [7] distributed with GCC provides much of the necessary information. The present work uses this information and puts it into the context of the ‘C6x’ architecture. Some details, however, remain unclear in [7] and will be investigated in this thesis by means of the GCC sources. Based on the results of this, it will be discussed which design options were available and which were chosen in implementing a ‘C6x’ backend. Finally, directions for future work will be shown.

Declaration of Authorship

I hereby declare that the whole of this diploma thesis is my own work, except where explicitly stated otherwise in the text or in the bibliography.

This work is submitted to Chemnitz University of Technology as a requirement for being awarded a diploma in Computer Science (“Diplom-Informatik”). I declare that it has not been submitted in whole, or in part, for any other degree.

Notational Conventions

In this work, the following notational conventions shall be used:

- Text passages that are meant to draw the reader's attention to them, even when only skimming over the text, will be typeset in **bold series**.
- Words to be emphasized, in a sense similar to emphasizing a word when reading a text out loud, will be written in *slanted shape*.
- C, RTL, and Assembler source code will be typeset in '`typewriter`' family, where :
  - Parts to be emphasized will be set in '**`bold`**' rather than '`medium`' series.
  - Placeholders will be set as in '`cmpgt` ⟨*reg*⟩`,` ⟨*reg*⟩`,` ⟨*pred_reg*⟩', where '⟨*reg*⟩' is a placeholder for a CPU register such as '`A5`', for example.
- In syntax rules, the following notation will be used:
  - terminal symbol: '**while**' (in bold)
  - non-terminal symbol: '*variable*'
  - optional part: '(*optional*)?'
- Filenames will be shown as 'somefile.c'. Unless specified otherwise, all filenames are meant as being relative to subdirectory 'gcc/' in the extracted GCC source tree.
- Occurrences of → 'keyword' in the text indicate that 'keyword' has an entry in the list of terms and abbreviations (Appendix A).

Contents

List of Figures

List of Tables

1. Introduction

1.1. Advantages of free Open Source Compilers

Today, many C compilers are available for a large variety of target platforms. However, not all of them are "free" in the sense of the definition maintained by the Free Software Foundation [3]. Free software has a number of advantages over commercial software:

- Researchers can publish their results without any restrictions imposed by non-disclosure agreements or copyright law.
- Patches can be distributed and applied freely by anyone interested in the field. This makes for higher efficiency of research, since any improvements are almost immediately available to the public.
- Research projects can build upon what is already available and do not have to start from scratch. In the field of compilers, this might be particularly useful when having devised a new optimization method which is to be tested in practice.
- Immediate reaction to bug reports is much more common with free software than with commercial software. Should the reaction take unacceptably long, only open source software allows the user to fix bugs himself.
- Financial means for license fees can be saved and put into additional manpower, instead.

1.2. Selection Criteria

That said, when looking for a compiler project to use as the basis for supporting a new target platform, it seems best to choose one that is free in the FSF sense. But there are more criteria to be taken into consideration:

- Number of target platforms already supported: The more processor types a compiler can already generate code for, the more likely it becomes that the features of a new target platform can be effectively mapped to the set of concepts supported by the compiler, making it unnecessary to implement the required concepts yourself.
- Number of input languages supported: New backend ports are, in principle, directly available to all the supported input languages of a compiler. The larger their number, the more people might find the port useful for their purposes.
- Ease of portability to new target platforms: Before having tried to port the backend to a new platform, it is hard to tell, whether porting will be relatively straight forward or rather involved. However, to get a first impression, one can look at the number of platforms already supported and at the size of source code that is specific to each of the targets. Both a high number of supported platforms and a small amount of source code per target indicate that porting should be feasible within a reasonable time.

- Ease of portability to new host platforms: This is difficult to assess for a compiler you are unfamiliar with. One possible option, though, is to look at how many host platforms the compiler currently can already be run on. The larger the qualitative variety of platforms, the more likely a good portability becomes.

1.3. List of Open Source Compiler Projects

When looking for C compilers that are available in the form of source code, the choice is limited to a relatively small number of projects. It follows a list of some of the subjectively more important ones:

GCC stands for "GNU Compiler Collection" [11], which aims at providing compilers for as large as possible a variety of input languages, code optimizations, and target platforms under a free software licence. In addition, the GCC documentation states that it is mostly platform independent, allowing it to be run on a notable number of "host platforms" [5].

GCC performs various optimization passes during compilation. This is the key to producing machine code that has an overall better run-time performance on many platforms than that of most other optimizing compilers. The prize to pay are higher computing power requirements for performing the compilation. Also, GCC needs to consider a lot of special cases in order to produce optimal code, requiring a large number of specialized routines dealing with them. So the source code of GCC must be rather large, which makes code maintenance somewhat more difficult.

lcc is a compiler project with focus on being easily retargetable. The authors take the approach of keeping their compiler small and simple, compiling notably more quickly than GCC, but with a less ambitious optimizer. As stated in [4], the target independent parts amount to $240,000$ lines of code in GCC, while lcc needs only $12,000$ lines (counting only the code generator for the Linux PC). The authors of the article put the difference between the two compilers like this: "GCC can emit better code and offers many options, which requires more source code, where lcc is easier to comprehend but is otherwise less ambitious."

lcc has a licence that permits to use it free of charge for non-commercial purposes. However, it is not free in the FSF sense. A good thing about lcc is the availability of a whole book dedicated to its implementation details. Less advantageous is the frontend of lcc, which is less flexible than the one of GCC and presumably makes support for other input languages relatively costly. The developers did not use lexer and parser generators like Lex and Yacc, but wrote both the lexical analyzer and the recursive descent parser by hand.

Watcom C/C++ and Fortran is a compiler suite originally developed by Sybase and now maintained by SciTech in the "Open Watcom" project [12]. Traditionally, it ran on MS-DOS, Windows, and OS/2 and also used to produce code for these platforms. Ports to Linux and BSD are currently being planned according to their "press releases" page, but seemingly have not been released as yet.

The Open Watcom project claims to have a code base of about 1 million lines under an Open Source licence approved by the Open Source Initiative (OSI). It should therefore be free in the FSF sense. However, no documentation seems to be available on the details of porting this compiler to new host platforms or targets.

TenDRA C compiler is a project [16] with the source code being available partly under BSD licence, partly under a somewhat free licence it got when being released by the Defence Evaluation and Research Agency (DERA) in 2001.

According to its web site, TenDRA focusses on code correctness rather than optimization. The compiler seems to support a number of platforms, but accepts only C as input language. No documentation seems to be available other than the comments in the source code. Also, traffic on the mailing list "tendra-announce@lists.tendra.org" has recently been very scarce, so that it is questionable whether the project is still being actively maintained.

1.4. Final Choice of the Compiler to use

None of the above compilers seemed to support 'C6x' as target architecture at the time of starting this work (and not even now). Being one of the most widely used compilers with support for a large number of target platforms, GCC appeared to offer the best prospects for being suitable as basis for implementing 'C6x' support. It also seemed to be the most promising candidate in terms of run-time performance of the generated target code due to its variety of code optimization techniques operating on the intermediate language level and therefore being usable with many different targets. Furthermore, compared to the other compiler projects, GCC offered the most comprehensive documentation for backend porters.

2. Fundamentals

The first part of this section gives an overview of the TI TMS320-C6000 architecture—‘C6x’, for short. In the second part, fundamental concepts of the GNU C compiler ‘cc1’ will be introduced, which are required in order to understand the later sections describing the implementation of a new ‘GCC’-backend for ‘C6x’.

2.1. Overview of the ‘C6x’ Architecture

This section gives a brief overview of the features of the C6x platform. A discussion of the architectural details (registers, pipeline, addressing, interrupts, instruction set, etc.) of the ‘C6x’ platform and its “generations” C62x, C64x, and C67x can be found in [14]. The following summary will focus on the features common to all three of these generations.

Knowledge of the architecture and the syntax of assembler statements is necessary in order to understand what kind of compiler output is appropriate in a particular situation. Porting the compiler to a new architecture involves specifying mappings from all elements of the C language to suitable assembler statements implementing the respective functionality. For example, ‘`if`’-statements are typically implemented using conditional branching. It is up to the compiler porter to specify how to put this into concrete assembler statements, such as ‘`[`⟨*condition register*⟩`] b` ⟨*label*⟩’.

2.1.1. Data Paths

The ‘C6x’ has 32 general purpose registers, each of 32 bit width. Consequently, most of the operations performed by the 8 functional units take 32 bit arguments and write 32 bit results. The **general purpose registers** are numbered A0 to A15 and B0 to B15, where the letter represents the data path this particular register belongs to: either A or B. The reason for making this distinction becomes apparent when taking a closer look at the **functional units** (.L1, .S1, .M1, .D1) and (.L2, .S2, .M2, .D2). The first group mostly works on data path A while the second group mostly works on data path B. For example, when unit .L1 is about to perform an ‘`add`’ instruction it must always write its result to one of the A registers and will normally obtain its arguments from register file A (except when using the cross path ‘1X’, which allows to access register file B from data path A). Similarly, .L2 would always write to and normally read from register file B, except when using ‘2X’. An illustration of the connection of functional units, register files, and cross paths to data paths can be found in Figure 2–1 in [14].

Also shown are the means of ‘C6x’ to access data in memory. Values can be loaded from memory via **load paths**: LD1 loads into register file A, LD2 loads into register file B. Storing register values to memory is done via **store paths** ST1 and ST2, for register file A and B, respectively. When using load or store paths, memory addresses must be sent to memory, first. This is the responsibility of the **data address paths** DA1 and DA2, where DA1 sends the address for loads performed via LD1 and for stores via ST1, and where DA2 sends the address for LD2 and ST2. As a convenience, when referring to the paths for memory access, the names T1 and T2 are introduced, where T1 stands for the tuple (DA1, ST1, LD1) and T2 for (DA2, ST2, LD2). These become important, for example, when an instruction shall be specified to access memory via T1, i.e. load/store in data path A, but to have its address calculated by functional unit .D2, which is part of data path B. This is possible, because each of the two data address paths are connected to both .D functional units, as can again be seen in Figure 2–1 in [14]. In order to specify this separation of address generation from the actual load/store one would write an assembler statement like the following:

‘`LDW .D2`**`T1`**` *B2(4),A5`’

The general format of an assembler instruction is given in section 2.1.3. The syntax of memory addresses is explained in 2.1.4.

2.1.2. Pipeline

As explained in chapter 6 of [14], the ‘C6x’ pipeline is divided into three successive “stages”, which are called “Fetch”, “Decode”, and “Execute”. Each ‘C6x’ instruction passes through these stages in the given order. Furthermore, the stages are subdivided into so-called “phases”, in which an instruction remains for exactly one CPU clock cycle before proceeding to the next phase. The phases are shown in Table 1.

Stage	**Phase**	
Fetch	PG:	Program address generate
	PS:	Program address send
	PW:	Program access ready wait
	PR:	Program fetch packet receive
Decode	DP:	Instruction dispatch
	DC:	Instruction decode
Execute	E1:	Execute 1
		⋮
	E5:	Execute 5

Table 1: ‘C6x’ Pipeline Stages and their Phases

The Fetch phases are responsible for obtaining instructions from memory. As long as there is no pipeline stall, one “fetch packet” of 8 instructions (32 bits each) is read in each clock cycle. Fetch packets containing multiple instructions are typical of very large instruction word (VLIW) architectures like the ‘C6x’.

On ‘C6x’, not all of the 8 instructions need to be executed in parallel, but they can be split into several “execute packets” to be executed sequentially. Whether or not this happens is determined by the p-bits in each fetch packet, which are analyzed in the **DP phase**. The details are explained in [14], section 3.5. Under normal CPU operation (no pending branches), a fetch packet remains in the DP phase until the last of its execution packets has been dispatched to the requested functional units for decoding. The pipeline phases prior to DP are stalled for this time. For each execution packet issued by the DP phase, the **DC phase** decodes the contained instructions for their respective functional unit. All the instructions are decoded in parallel and in one clock cycle.

Afterwards, the instructions enter their **E1 phase**. Many instructions do not need any more phases other than E1 and are therefore called “single-cycle instructions”. A special advantage of them is the fact that the instructions in the next execute packet can already use their results as arguments. Instructions requiring multiple execution cycles are posing the problem of the result being written much later to the register files. For instance, the ‘`ldw`’ instruction (load word from memory) needs all 5 execute phases E1 to E5 until the data from memory is finally written to the register files. At the time, the next execute packet EP is in E1, ‘`ldw`’ is only in E2 and its result is far from being available yet. If, for example, EP contains an ‘`add`’ (add two registers into a third one) with the result register of the ‘`ldw`’ instruction as an argument, it will see the register’s old value, which might not be intended. However, if the ‘`add`’ was not issued one, but 5 cycles after ‘`ldw`’, ‘`ldw`’ would just have finished E5 at the time of ‘`add`’ entering E1. In this case, the result of ‘`ldw`’ would just be available to the ‘`add`’ instruction. The 4 cycle gap that has to be inserted between the two instructions in order to have the result of ‘`ldw`’ available to ‘`add`’ is referred to as “delay slots”. One has to keep this in mind when implementing the specification that tells the compiler how to generate assembler code. Other instances, where delay cycles are important, are branch instructions (5 delay slots until fetch packet at target is executed) and multiply operations, as explained in more detail in [14], section 6.2.

2.1.3. Format of Assembler Source Statements

As described in [13], Section 3.5, the general syntax of an assembler source statement is as follows (terminal symbols shown bold, here):

(*label* (**:**)?)? (**||**)? (**[** (**!**)?*condReg***]**)? *mnemonic* (*unit spec*)? (*operand list*)? (**;** *comment*)?

Discussing the details of this would go beyond the scope of this work. However, an example of a valid assembler statement shall be discussed briefly:

```
label0:  [!B0] MVKL .S2 3,B1 ; loading immediate into B1
```

The syntactic parts of this statement correspond to the non-terminal symbols of the syntax definition from above as follows:

Non-Terminal Symbol	**Part of the Example**	**Meaning**
label	`label0:`	• a code label that can be jumped to
[(**!**)?*condReg***]**	`[!B0]`	• perform instruction only if B0 = 0
unit specifier	`.S2`	• the functional unit to use
operand list	`3,B1`	• source op.: 3, target op.: B1
; *comment*	`; loading ...`	• asm source documentation

As a convention, an ‘*operand list*’ starts with the source arguments and ends in the target argument, e.g. as in: ‘`ADD` ⟨*src1*⟩`,`⟨ *src2*⟩`,`⟨ *dst*⟩’. Note that not all functional units can accept all types of instructions. Instead, they are specialized in performing a certain kind of operation. The .M1 and .M2 units, for example, can be used for various kinds of multiplications but hardly for anything else. Table 3–2 in [14] lists the names of the instructions each unit is able to handle. If the '*unit specifier*' is omitted, the assembler will resort to adding itself a suitable unit specification to the binary instruction word it generates.

Not shown in the example above was the '||' feature, which can be used to combine the current statement with preceding statement(s) into an execution packet to be executed in parallel. However, if not done with care, this may violate concurrent CPU resource usage constraints of the C6x architecture. Hence, in order to prevent conflicts, any compiler utilizing parallel execution must keep track of which CPU resources its generated assembler instructions allocate and at which processor cycles.

2.1.4. Memory Addressing Syntax for Load/Store Instructions

There are a number of ways in which to specify the memory address to store a register value to ('`stw`', '`sth`', '`stb`') or load data from ('`ldw`', '`ldh`'('`u`'), '`ldb`'('`u`')). Memory addresses are always relative to some register R_{base}. For R_{base}='`A4`', the address would be '`*A4`'. This basic addressing type is called 'register indirect' in section 3.8.3 of [14].

As an option, the base register can be offset either by a register R_{offs} (from the same register file as R_{base}) or a constant 5-bit unsigned value (see [14] for more), which is subtracted from or added to R_{base}, e.g.: '`*-A4[24]`' or '`*+A4[A5]`'. This is called 'register relative' addressing for immediate offsets and 'base + index' addressing for register offsets. The offsets within '`[...]`' are scaled according to the width of the instruction in which they are used. For example, '`ldw`' (load word) scales address offsets by 4. With immediate offsets, parentheses can be used instead of brackets in order to prevent the scaling, e.g. as in '`*-A4(24)`'.

The addressing type just described computes an address at some offset from R_{base}, but does not actually modify R_{base} itself. However, 'C6x' offers the possibility of not only computing the target address, but also assigning it to R_{base}. Firstly, there is a *pre*-increment/decrement mode, e.g. '`*++A4[24]`', where the computed address is both assigned to R_{base} and accessed in memory. Secondly, there is a *post*-increment/decrement mode, e.g. '`*A4++[24]`', where the computed address is assigned to R_{base} only *after* accessing memory at the old value of R_{base}.

2.2. GCC Toolchain

The term "toolchain" is used to describe the sequence of tools that is typically used in order to translate a collection of C source code files to an executable file, which can directly be run by the operating system on the target platform.

With the GNU Compiler Collection, the toolchain is controlled by "gcc", the so-called "compiler driver". 'gcc' builds a pipeline of tools, which it uses to process the C sources all the way through to the final executable. This pipeline is shown in Figure 1.

First of all, 'gcc' passes all C source files to be processed to "cc1", the GNU C compiler. '**cc1**' preprocesses each file, following any directives such as '`#define`' and '`#include`', for example, and expanding any macros it encounters. The resulting tokens are then passed on to a

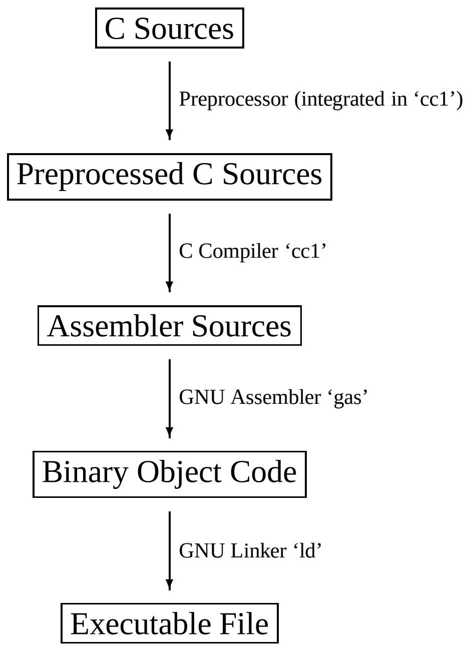

Figure 1: Toolchain of GCC

Yacc-generated parser, which recognizes all the elements of the C language and “reduces” tokens into non-terminal symbols according to the syntax rules defined in file ‘c-parse.y’. Whenever the parser is able to reduce the definition of a function F to an ‘`fndef`’ non-terminal, it calls ‘`finish_function`’ (‘c-decl.c’) to complete the compilation of F, which eventually results in outputting assembler statements for this function. After this, the parser resumes its work and processes any remaining C-functions in the current source file. When finished with a source file, control is returned to ‘gcc’, which passes the generated assembler output to ‘gas’ for further processing.

‘**gas**’ translates the incoming assembler statements into relocatable binary object code for ‘C6x’. In order to be able to do this, ‘gas’ would have to be ported to this target platform and to be configured to actually use it. ‘gas’ makes use of the Binary Format Descriptor library (BFD) (part of the ‘binutils’ collection [6]) in order to output any relocations in compliance with the specifications of the desired binary object format.

When this work is done, the GNU linker ‘**ld**’ combines the object files generated from the different C source code modules into the final executable. Again, the BFD library is used for handling symbol relocations.

Knowing the structure of the toolchain, the most important steps needed in porting GCC to the ‘C6x’ target platform can be summarized as follows:

- add support for ‘C6x’ to ‘cc1’
- add support for ‘C6x’ to ‘gas’
- add support for the desired binary object format to the BFD library

This work will focus on ‘cc1’. Writing ports for ‘gas’ and the BFD library would be separate projects, which are not the subject of this work. The same applies to the other tools in ‘binutils’, which may also need to be updated in order to provide ‘C6x’-related functions.

Another aspect that might need to be considered is the configuration of the parts of the toolchain. It is, in most part, controlled by the file ‘specs’, which follows the syntax of the “Specs Language” as documented in a comment in ‘gcc.c’. The compiler driver reads this file to learn the command line options that have to be used for invoking the different parts of the toolchain. Target-specific modifications can be made by providing definitions for some of the ‘∗_SPEC’-macros in the machine description header file ‘config/c6x/c6x.h’. The name and a short description of most of these macros is given in [7], node ‘Driver’.

2.3. C compiler ‘cc1’

The purpose of this section is to introduce fundamental concepts of the C compiler ‘cc1’, which is part of the GNU Compiler Collection [11]. A profound understanding of these concepts can help when working through the details of the implementation of a new backend for the ‘C6x’ architecture, which will be discussed in Chapter 3.

2.3.1. Introduction of Important Terms

This section will introduce some important terms of GCC and should be read before proceeding to the following sections for better understanding.

Intermediate Code—RTL When translating a C source code module, GCC is not able to generate assembler code directly but needs to represent the code in an intermediate language, called Register Transfer Language (RTL). The RTL is processed by several compiler passes and finally translated into assembler code. Some of these passes make modifications so that the final output will be valid code, others perform optimizations with respect to later runtime performance. Some more comments on RTL and an example of typical RTL expressions (also known as RTX) can be found in Section 2.3.2, on page 13.

Register Allocation The number of variables a C function can use in its computations is potentially infinite. The number of CPU registers available on a target machine is limited, however. Still, the compiler must somehow map the variables to registers in order to generate valid assembler code. One solution to this problem is to put all variables into memory, and to load them only when needed. Computations can then be performed in registers and the results must immediately be stored back to memory. This method requires only very few CPU registers, but has the major drawback of not using the performance potential that lies in the large number of CPU registers offered by most RISC machines. So, a better approach might be to hold as many C variables as possible in registers rather than memory in order to save a lot of load and store operations. When using this method, the compiler has to keep track of which CPU registers are free and which are allocated at any point of the program. This information can then be used to assign a CPU register to each C variable, if possible. This is called ‘register allocation’.

Pseudo Registers and Hard Registers Intermediate code generation from a C source is a rather involved task already in itself. If register allocation had to be implemented as part of it, very complex code would probably result. ‘Pseudo registers’ serve as an interface here, allowing to implement these tasks in two separate passes. The code generation pass may allocate a potentially infinite number of pseudo registers and generate its intermediate code (RTL) without any restrictions in this respect. Of course, these registers have to be mapped to *real* CPU registers, called ‘hard registers’, before outputting any assembler code. This is done by the register allocator, which initializes the ‘`reg_renumber`’ array in a second pass. The intermediate language in GCC uses the same representation for both pseudo and hard registers. The only difference is the numbering scheme. For hard registers, numbers start at zero and count upward to one less than the number of available CPU registers on the target. The numbers of pseudo registers start immediately afterwards and count upward as they are allocated.

Note that the register allocator is only responsible for initializing ‘`reg_renumber`’. A separate task, however, is to modify the actual register references in the RTL code to reflect this allocation. This is done in the ‘reload’ pass (to be explained throughout the following sections) by calling function ‘`alter_reg`’.

Virtual Registers and Register Instantiation The first few pseudo register numbers are reserved for the so-called ‘virtual registers’. Like all pseudos, virtual registers do not directly correspond to existing CPU registers, but are virtual and get substituted by hard registers as soon as the compiler has gathered enough information to do so. In the case of virtual registers, there may also be a sum of a hard register and some offset substituted for them, which in fact is the most common case. For virtual registers, the substitution process is called register instantiation and performed by function ‘`instantiate_virtual_regs`’.

Unlike normal pseudos, virtual registers are not meant to be allocated when a register for a local or temporary variable is requested, but are preallocated during the initialization phase of the compiler. Their purpose is to serve as address to locations on the stack for which an absolute address cannot be computed yet, because it might be subject to change as the compilation progresses.

For example, when a location on the stack needs to be addressed and its offset from the stack pointer will later definitely be a constant but cannot be determined at the current stage of compilation, a virtual register would be used to represent the address. This can happen, for example, when the block of local and temporary variables for some C function resides between the stack pointer and the location to be addressed, as shown in Figure 2. The offset of this location to the stack pointer cannot be determined, yet, because the number of bytes required for temporary variables can grow until *all* of the current C function has been translated into RTL. So, a virtual register is defined to point “just after the variable block” and any affected memory addressing expressions are generated relative to this virtual register.

Basic Blocks In [1], p. 528, the following definition is given: “A ‘basic block’ is a sequence of consecutive statements in which flow of control enters at the beginning and leaves at the end without halt or possibility of branching except at the end.” A typical example of a basic block

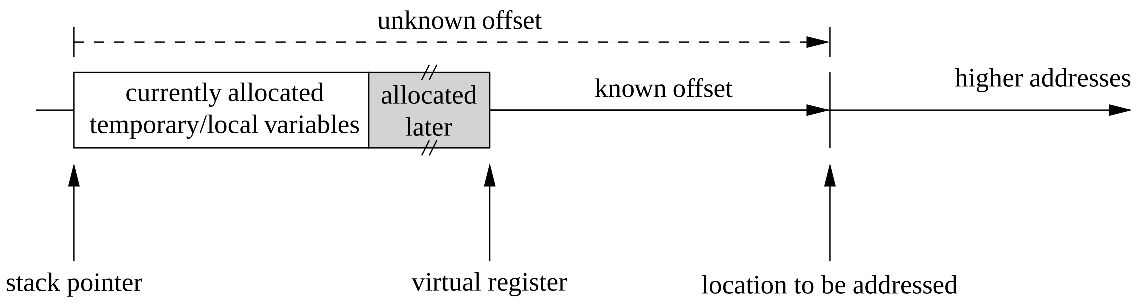

Figure 2: Example of Virtual Register Usage

would be a sequence of simple arithmetic computations, where the block does not contain any labels for jumping into it, nor does it contain any opportunity of jumping out of it. Hence, a basic block is something that always runs in whole from top to bottom, never only part of it. The concept of basic blocks appears in most of the RTL-processing algorithms of GCC. For example, register allocation of pseudo registers to hard registers is divided into two algorithms, one of which handles allocation inside basic blocks, called ‘local register allocation’, and the other handles allocation on the global level (‘global register allocation’), i.e. taking into account the control flow between the basic blocks.

2.3.2. Stages of the Compiler

The compilation process from C to assembler code can be seen as a sequence of three stages, which are called ‘Frontend’, ‘Middle-End’, and ‘Backend’. The tasks of these stages shall be demonstrated by means of a short example. Consider the following C program:

```
void main()
{
    int a=1, b=2;

    a = a + b;
    return;
}
```

We assume that all compiler optimizations are turned off and that therefore the framed computation will be performed even though its result is never used. The following paragraphs will show the processing, which the framed statement undergoes on its way to a final assembler statement.

Front-End First, the parser is run in order to build up a tree representation of the sample C statement. Since it can only process readily recognized tokens, the parser calls the lexer in order to have it split up the input. Each time the lexer is called, it returns one token: ‘`a`’, ‘`=`’, ‘`a`’, ‘`+`’, ‘`b`’, and finally ‘`;`’. Actually, the lexer does not return these character representations, but returns the token types and the semantic values of the tokens.

The parser algorithm uses this data for building up the parse tree according to the syntax rules of the C language. The parse tree for the sample statement is shown in Figure 3. Each box represents one data structure of type ‘`union tree_node`’, which can describe either a basic or a compound element of the C language. All components of such a union are ‘`struct`’s, each containing a ‘`struct tree_common`’ at the beginning. This makes sure that all tree nodes have a ‘`code`’ field at the same memory offset. The ‘`code`’ field is of type ‘`enum tree_code`’ and indicates which kind of C language element this node represents. It is shown topmost in each box of the Figure. The other fields shown in some of the boxes are ‘`name`’ and ‘`type`’, of which the latter applies to all kinds of C expressions and describes their C types. The former applies to ‘`_DECL`’ nodes such as those for the variables ‘`a`’ and ‘`b`’. It indicates the name that was given to the declared element in the input source code. Expression nodes can have operands, allowing for complex expressions to describe the parts they are composed of. The ‘`PLUS_EXPR`’, for example, has two of them, representing the addends. Pointers to these operands are stored in a ‘`tree_node`’ substructure field, which can be accessed by macro ‘`TREE_OPERAND(`⟨*node*⟩`,` ⟨*opnum*⟩`)`’. In the figure, the pointers are shown as arrows.

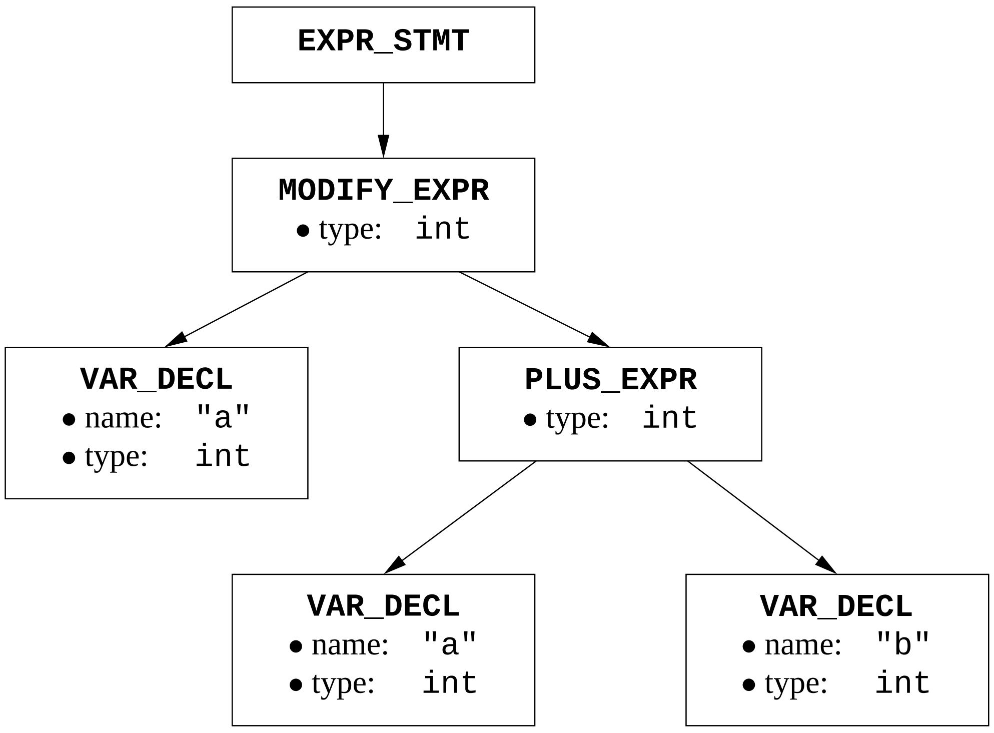

Figure 3: Parse Tree for ‘`a = a + b;`’

When the parse tree for one C function has been completed, it gets translated into Register Transfer Language (RTL). This process is called ‘Expansion’, because the structured tree representation is *expanded* to a more or less linear list of actions that correspond to the assembler statements to be output after further processing.

RTL e**x**pressions (RTX) are similar to Lisp expressions. They exist in two forms: internal and textual. What you see and type is the widely known textual form with its large number of parentheses, whereas the internal form is a linked tree, representing the same structure as the textual form. The internal form is better suited for being processed and modified by algorithms, because its parts can be accessed in a much more direct fashion. Almost all of the passes of the compiler therefore work on internal RTL rather than on textual RTL and textual assembler statements. The main advantage of RTL, however, is its abstractness. In RTL, operations are described mostly independent of concrete assembler statements and independent of the target architecture. This allows for many compiler algorithms to work on different kinds of target architectures without the need to be modified. Examples hereof would be register allocation, jump optimization, and loop optimization. It would be tedious to implement slightly modified versions of these algorithms over and over again. The details of RTL are explained in [7], node 'RTL'. Therefore, the following paragraph will provide only a short introduction to RTL code.

Let us consider again the parse tree from Figure 3. The textual form of its translation into RTL would look similar to the following:

```
(set (reg:SI 7) (mem:SI ⟨address of 'a'⟩))
(set (reg:SI 8) (mem:SI ⟨address of 'b'⟩))
(set (reg:SI 7) (plus:SI (reg:SI 7) (reg:SI 8)))
(set (mem:SI ⟨address of 'a'⟩) (reg:SI 7))
```

The first RTL statement is a '`set`' operation, which represents an assignment from the second argument to the first argument. The '`mem`' stands for the value in memory at the address of variable '`a`'. The '`reg`' refers to register 7 on the target machine. So the assignment actually is a 'load' from memory. It might later be output as the following assembler statement: '`ldw *A10, A7`', where '`A10`' would have to hold the address of '`a`'. The width of the data transfer is determined by '`:SI`', which is short for 'Single Integer' and always means '4 byte' within GCC. The second RTL statement is completely analogous to the first one, loading variable '`b`' into register 8. The third statement is again a '`set`' operation, but this time the source argument is a '`plus`', representing the sum of '`A7`' and '`A8`'. The result is written to the target argument, which is register '`A7`'. So the corresponding assembler statement to be output will be: '`add A7, A8, A7`'. The fourth statement finally writes the result of the addition, which is now in '`A7`', back to variable '`a`' in memory.

In this brief example, only few RTL statements could be introduced. There exist many more of them, which can be used for describing a wide range of arithmetic operations, jumps, labels, function calls, etc. For details, please refer to [7], node 'RTL'.

Note that, at the time of expanding the tree representation to RTL code, the compiler ideally should not need any knowledge of the concrete assembler statements for loading, storing, or adding, because everything would be expressed in abstract RTL. Unfortunately, it turns out that this does not work in practice, because most of the architectures offer only a small subset of the operations that are describable in RTL. So GCC is not free in which RTL statements it may generate, but must abide by the constraints imposed by the target architecture currently in effect. If, for example, an architecture is incapable of adding in SI-mode (32 bit), but knows how to add in HI-mode (16 bit), GCC can emit two HI-additions in order to emulate an SI-addition. In order to know that no 32 bit addition is available, the compiler needs to be made aware of the capabilities of the target machine. This can be achieved by implementing a 'machine description', which includes a definition file for all available instruction patterns ('config/⟨*arch*⟩/⟨*arch*⟩.md') and one for certain meta information such as the number of available registers ('config/⟨*arch*⟩/⟨*arch*⟩.h'). More information on machine descriptions can be found in Section 2.3.8.

Middle-End Whenever function '`c_expand_body`' has finished expanding the tree representation of a C function into RTL code, it runs the Middle-End, which is implemented by function '`rest_of_compilation`'. The Middle-End is responsible for any compiler passes that take RTL as input, modify it in ways specific to the current pass, and produce RTL as output.

Most of them are *optimization* passes for improving runtime performance of the assembler output. But there are also some passes, which are *essential* for generating valid assembler code. The latter include the 'Reload' pass as well as the local and global 'Register Allocation' pass. At the current point, too little terminology has been introduced to explain these passes further, but later sections will catch up on this. An overview of important compiler passes is given in [7], node 'Passes'.

Backend When the Middle-End has finished all its passes it invokes the Backend (function '`final`'), which is also called 'final pass', because it performs the *final* translation from RTL code to assembler statements. At this stage, all pseudo-registers must have been allocated to hard registers and any intermediary RTL code must have been removed. For all remaining RTL statements there must be a matching 'instruction pattern', which defines the translation to a corresponding assembler statement. Such patterns must be defined as part of the machine description in file 'config/⟨*arch*⟩/⟨*arch*⟩.md'. For example, for the RTL statement

```
(set (reg:SI 6) (plus:SI (reg:SI 7) (reg:SI 8)))
```

there must be a matching RTL template, which might look as follows:

```
(define_insn "addsi3"
    [(set
        ⟨template variable 0⟩
        (plus:SI
            ⟨template variable 1⟩
            ⟨template variable 2⟩))
    ]
    "⟨condition⟩"
    "add %1, %2, %0")
```

Template variable 0 will be instantiated to '`(reg:SI 6)`', variable 1 to '`(reg:SI 7)`', and variable 2 to '`(reg:SI 8)`'. The assembler output template contains references to these variables and gets instantiated accordingly, producing the output '`add A7, A8, A6`'. More details on instruction patterns and template variables will be given in Section 2.3.9 and 2.3.10.

2.3.3. Initialization

The first action, when ‘cc1’ starts, is to do some basic initialization of certain integral parts of the compiler (function ‘`general_init`’), for example the garbage collector (via ‘`init_ggc`’) and the identifier hash table (via ‘`init_stringpool`’) that will later hold information about user- and pre-defined C identifiers.

Once these parts are set up and command line options have been parsed (by ‘`independent_decode_option`’) and processed (by ‘`*lang_hooks.post_options`’ and ‘`process_options`’), the next step is to allocate initial data structures for some of the modules implementing the compiler backend passes (function ‘`backend_init`’). For example, the loop optimization pass (‘`init_loop`’) and the reload pass (‘`init_reload`’) get initialized here. Also, some of the time costs caused by performing certain operations on the target machine are computed (function ‘`init_expmed`’) and stored in variables for later use by ‘expmed.c’ and others. Such information can be useful when the compiler has two or more options of performing a certain kind of computation and needs to decide which one to actually generate code for on a specific target platform. There are a lot more target-dependent initializations in function ‘`backend_init`’, which cannot be explored, here, in detail. It is just important to note that the main aim of ‘`backend_init`’ is to do any initializations somehow dependent on a particular target machine. One of these is to determine the individual capabilities of the machine, for example.

The next step in function ‘`do_compile`’ is to call function ‘`lang_dependent_init`’ in order to perform all initializations specific to ‘C’ as the input programming language. Among other things, the latter function opens the assembler output file for writing (function ‘`init_asm_output`’) and initializes the debugging information generator facilities by calling ‘`*debug_hooks->init`’. It also invokes the callback function pointed to by ‘`*lang_hooks.init`’, which is ‘`c_objc_common_init`’ for the ‘C’ language, and which in turn calls ‘`c_init_decl_processing`’ and ‘`c_common_init`’. The first of the two functions creates and registers type nodes for scalar types that are supposed to be “predefined” from the perspective of the ‘cc1’ user. Examples would be ‘`long unsigned int`’ and ‘`char`’. It also registers the builtin functions defined in ‘builtins.def’, e.g. ‘`__builtin_alloca`’ and ‘`__builtin_va_start`’. The second of the two functions, ‘`c_common_init`’, mainly sets up the C lexer, preparing it for tokenizing input from the main source file given on the command line. To this end, it calls ‘`init_c_lex`’, which opens the input file for reading and registers handlers for directives like ‘`#define`’ and ‘`#include`’ (function ‘`_cpp_init_directives`’). It also initializes the preprocessor identifier hash table ‘`parse_in->hash_table`’ to the value of the global variable ‘`ident_hash`’, which was already initialized by ‘`init_stringpool`’, mentioned above, to point to the C identifier hash table. Hence, there is only *one* hash table, which is used for both C identifiers and preprocessor identifiers, but *two* separate mechanisms for accessing it: C identifiers are looked up and inserted by function ‘`get_identifier`’, whereas the same is done for preprocessor identifiers by function ‘`cpp_lookup`’.

This overview of the initialization phase is far from being complete. However, it does not aim at covering every single detail, but wants to provide some starting points for further exploration of the source code.

2.3.4. Lexing

Lexing is actually not a pass of the compiler, but more like a module, which is called multiple times during compilation (via macro ‘`YYLEX`’), whenever the parser (function ‘`yyparse`’) needs to know the next input token. ‘`YYLEX`’ is defined to perform a call to function ‘`yylex`’, which is a thin wrapper for function ‘`_yylex`’. The main tasks of ‘`_yylex`’ are: concatenating string literals, looking up identifiers in order to decide whether they stand for a type name (‘`TYPENAME`’) or a variable/function identifier (‘`IDENTIFIER`’)—this distinction is important for preventing ambiguities during parsing—and translating token types from their representation in type ‘`enum cpp_ttype`’ to the terminal symbol encoding used by ‘c-parse.y’. For actually obtaining the next token from the input stream, however, ‘`_yylex`’ relies on the functionality provided by function ‘`c_lex`’.

‘`c_lex`’ is responsible for extracting the semantic information carried by each token. For instance, while the token type for the number literal ‘`32`’ would be ‘`CPP_NUMBER`’, its semantic information would be the value 32, seen as an integer. ‘`c_lex`’ would put this information in a ‘`tree_node`’ union and return it as the token’s ‘semantic value’. Semantic values are a basic concept of the Bison parser generator [9] and are further processed by the definitions in the Bison grammar file ‘c-parse.y’. Please refer to Section 2.3.5 for more on this.

In order to fulfil its tasks, ‘`c_lex`’ calls several functions: ‘`cpp_get_token`’ to obtain the next token as a ‘`struct cpp_token`’, and functions like ‘`interpret_integer`’, ‘`lex_charconst`’, and ‘`lex_string`’ in order to extract the semantic value of the token. Function ‘`cpp_get_token`’ wraps up the core functionality of the lexer, which comprises the points listed in Table 2.

There is special handling for words in the input that look like C identifiers (starting with a character and consisting of alpha-numeric characters). This is because the lexer cannot immediately tell whether it is dealing with a user-defined preprocessor macro, which would have to be expanded, or a user-defined C identifier for a variable, type, or function, which would have to be passed through unmodified. These cases can be distinguished by looking up the word in the identifier hash table. The corresponding call chain is as follows:

$$\begin{aligned}
&\text{‘}\texttt{cpp\_get\_token}\text{’}\\
\xrightarrow{\text{calls}}\ &\text{‘}\texttt{\_cpp\_lex\_token}\text{’}\\
\xrightarrow{\text{calls}}\ &\text{‘}\texttt{\_cpp\_lex\_direct}\text{’}\\
\xrightarrow{\text{calls}}\ &\text{‘}\texttt{parse\_identifier}\text{’}\\
\xrightarrow{\text{calls}}\ &\text{‘}\texttt{ht\_lookup}\text{’}
\end{aligned}$$

If the ‘`type`’ field of the ‘`cpp_hashnode`’, which is returned, is ‘`NT_MACRO`’, the preprocessor itself tries to expand it. Otherwise, the hash node, which probably holds the declaration information of some C identifier, is returned unmodified.

This quick overview of the ‘cc1’ was written with the aim in mind of providing some entry points for further code analysis of the lexer. Normally, when just implementing a backend for a new target architecture, one would assume to be able to do without any knowledge of the lexer. While this may be true in some cases, there are a number of occasions where some knowledge of lexer details proves useful.

	Functionality	Implementing C-function
1.	handling of directives (‘`#define`’, ‘`#include`’, etc.)	‘`cpp_get_token`’ $\xrightarrow{\text{calls}}$ ‘`_cpp_lex_token`’ $\xrightarrow{\text{calls}}$ **‘`_cpp_handle_directive`’**
2.	expansion of macros	‘`cpp_get_token`’ $\xrightarrow{\text{calls}}$ **‘`enter_macro_context`’**
3.	classification of number/string literals and recognition of operators	‘`cpp_get_token`’ $\xrightarrow{\text{calls}}$ ‘`_cpp_lex_token`’ $\xrightarrow{\text{calls}}$ **‘`_cpp_lex_direct`’**

Table 2: Essential Functions of the Lexer

2.3.5. Parsing

GCC uses the Bison parser generator [9] for generating C code that parses the currently configured input language. When building GCC with C as the input language, the name of the Bison grammar file is ‘c-parse.y’ [1]. The parsers generated by Bison are ‘bottom-up’ parsers, which means that parsing proceeds by shifting tokens onto a stack called ‘state stack’, while looking for matching grammar rules, which allow a reduction of a group of symbols on the top of the stack to some non-terminal symbol. The symbols are popped off and the new symbol is pushed onto the stack, where it may be part of further reductions. For syntactically correct C programs, the last step in this process is a reduction to the start-symbol of the C grammar, which is the last to remain on the state stack.

In parallel to the state stack ‘`yyss`’, a semantic value stack ‘`yyvs`’ is maintained by the parser function ‘`yyparse`’. Whenever a reduction occurs, the semantic action (see [10], node ‘Semantic Actions’) of the reducing grammar rule (see [10], node ‘Rules’) will be executed in order to handle the reduction in terms of semantic values. For example, please consider the following grammar rule, which was taken from ‘c-parse.y’ and slightly adapted:

```
expr_no_commas:
    expr_no_commas '+' expr_no_commas
        { $$ = parser_build_binary_op ($2, $1, $3); }
```

When a reduction by this rule occurs, two C expressions connected by a ‘`+`’ operator have just been recognized. The parse trees, which have been built up for the two expressions, are now their semantic values. In the semantic action, they can be referenced by writing ‘`$1`’ and ‘`$3`’. ‘`$2`’ refers to the semantic value of the ‘`+`’ operator, as initialized by function ‘`_yylex`’. While the

[1]‘c-parse.y’ is automatically generated. The hand-coded version is ‘c-parse.in’, which gets processed by ‘Makefile’.

state stack is updated automatically by the parser, the new value to be pushed onto the *value stack* must[2] be specified explicitly in a semantic action, which is done by writing an assignment to ‘`$$`’. The function ‘`parser_build_binary_op`’ is responsible for performing any actions required to combine the two parse trees into a new one. This is quite complex, since it involves tasks such as type checking, implicit type conversion, and constant folding.

In the first place, semantic values are created (or looked up in the identifier hash table) by the lexer. For each token it recognizes, an appropriate semantic value is put into the global variable ‘`yylval`’, which is pushed onto the value stack by the parser when the token is being ‘shifted’ (see [10], node ‘Algorithm’).

One of the most important elements of the C language is the notion of a ‘function’. Functions can be used by the user to express complex computations, which are to be performed when he runs the binary that the compiler makes from his source code. On the compiler side, function definitions must therefore have the effect of generating a block of assembler instructions implementing the function. In the parser module ‘c-parse.y’, function definitions are represented by the non-terminal symbol ‘`fndef`’. The semantic actions of the grammar rule describing the ‘`fndef`’ reduction contain statements that initiate the further compilation of the function. The first of these statements, placed in a mid-rule action (see [10], node ‘Mid-Rule Actions’), is a call to ‘`start_function`’, which is made before the body of the user function has even been seen by the parser. This is necessary in order for local definitions in the function body to be handled correctly. When all of the function has been seen, the end-of-rule statements of the ‘`fndef`’ are executed in order to initiate its further processing. To this end, they contain a call to ‘`finish_function`’, which calls ‘`c_expand_body`’ in order to have the statements in the function body expanded to RTL and to have the rest of the compilation carried out all the way to final assembler output.

2.3.6. Generating RTL

When the parser has done its work, building up the parse tree for a C function, the statements in the function have to be translated into RTL for further optimization and processing. This is called ‘RTL generation pass’ (see [7], node ‘Passes’ at ‘RTL generation’). RTL generation, which is also called ‘expansion’, is controlled by function ‘`c_expand_body`’, which has several tasks:

Firstly, it is responsible for setting up default values for variables which describe various properties of the current function and which will receive their final values during the compilation passes to follow. The main initialization work is done in ‘`prepare_function_start`’ (called by ‘`init_function_start`’) and its helper functions. A ‘`struct function`’ and its contained structs are allocated to hold properties of the function and various status information with respect to scope, to ‘`if`’, ‘`case`’, and iteration statement blocks, and much more. Also, the emission of RTL code is prepared by calling ‘`init_emit`’. This initializes a double-linked list of ‘`insn`’ RTX and allows for the ‘`emit_*`’ functions in ‘emit-rtl.c’ to be legally used.

Secondly, ‘`c_expand_body`’ calls ‘`expand_function_start`’, which, among other

[2]If no semantic action is specifi ed, Bison will provide a default action, but this is usually not intended.

things, initializes RTX for the 'return_label' and for the incoming parameters ('assign_parms'). For nested functions (a GCC extension), it also emits RTL for copying the static chain pointer register to its stack slot and creating 'context_display', a chained list of tree nodes, which hold the static links for the enclosing functions. A 'NOTE_INSN_FUNCTION_BEG' is emitted in order to mark the beginning of the function body for use by, for example, 'init_alias_analysis' (alias analysis) and 'final' (for debugging output generation).

Third comes the main step of the RTL generation pass, which is to expand all C statements of the current function. To this end, 'c_expand_body' calls 'expand_stmt', which obtains the list of statements from 'DECL_SAVED_TREE'. This is a 'tree list', which the parser built up using 'add_stmt' to add new statements. The variable 'last_tree' is used by 'add_stmt' as a pointer to the most recently added statement. The start node of the list is set up by 'store_parm_decls'$\xrightarrow{\text{calls}}$'begin_stmt_tree' to be '&DECL_SAVED_TREE (current_function_decl)'.

Fourth and finally, 'expand_function_end' is called in order to emit anything requiring information, which cannot be determined until after the function has been completely expanded. An example would be the question whether the function contains a call to 'alloca' (dynamically allocates automatic storage on stack) anywhere, and therefore, at least on some machines, needs code to save the stack pointer on function entry and restore it before the function returns. If so, appropriate statements have to be inserted at the beginning and at the end of the already emitted RTL.

Now, we shall take a closer look at 'expand_stmt' already mentioned in the third step. The task of this function is to go through the 'tree list' it is passed and emit RTL for each '∗_STMT' node of this list. For example, when a 'SWITCH_STMT' is seen, 'genrtl_switch_stmt' will be called and will expand the switch statement to RTL, which includes emitting statements for evaluating the switch expression, statements representing the body of the switch statement, and statements that conditionally jump to the correct case label as determined by the value of the switch expression. To find information on where 'SWITCH_STMT' tree nodes are created, it may help to look into the file 'c-parse.y' at 'select_or_iter_stmt:'. The precise structure of a concrete '∗_STMT' node can be examined with the help of 'gdb', which allows you to set a breakpoint to 'expand_stmt' and do a 'call debug_tree (t)'. This will call function 'debug_tree' defined in 'print-tree.c', which will recursively—at least to some depth—print the fields of the tree node 't' to 'stderr'.

There are a lot more aspects to RTL generation than those described in this section. For example, there is a large module named 'expr.c' ($\approx 11,000$ lines), which is only concerned with the expansion of all kinds of C expressions. The entry point for this would be function 'expand_expr', which uses a lot of helper functions from other source modules. Arithmetic expressions, for example, require the help of 'expand_unop', 'expand_binop', and other functions from 'optabs.c'.

Another relatively complex aspect are function calls. Calls are represented as 'CALL_EXPR' nodes. They normally occur as part of expression evaluation and are therefore handled by 'expand_expr', which delegates their expansion to 'expand_call' in file 'calls.c'. As GCC supports a large variety of target machines, the code for expanding calls has to be flexible in terms of how to pass function arguments (on stack or in registers, in preallocated or in pushed stack area, in normal or in reverse order) and how to share the work of setting up the stack frame between caller and callee.

When ‘`c_expand_body`’ has finished expanding the tree representation of the current function, it calls ‘`rest_of_compilation`’ in order to initiate further processing, which means handover from the frontend to the middle-end. The middle-end optimizes the RTL code and prepares it to be passed to the backend, so that valid assembler code can be generated. Most importantly, this involves a register allocation and a reload pass. As mentioned before, register allocation assigns a hard reg to each pseudo register and reload modifies the RTL code accordingly. All pseudos are turned into either hard registers or memory locations on the stack. Another task of reload is to ensure that references to memory get reloaded to registers, when the matching instruction pattern in the machine description has constraints that require this.

2.3.7. Final Pass

So far, the C source code has undergone a number of processing steps: In the frontend, it has been lexed, parsed and expanded to RTL code on a per-function basis. In the middle-end, the RTL code has been subjected to register allocation, reloading, and a number of optimization passes (see [7], node ‘Passes’), where instances of abstract RTL have been translated to a form that is closer to the target machine, and that is free from pseudo registers, virtual registers, operands with unmet ‘constraints’ (to be explained in Section 2.3.10), and other elements preventing a direct translation to valid assembler code. Now, one final step remains to be done, which is the pattern-based translation of the RTL instructions to assembler statements. This is called ‘final pass’.

As the name suggests, the final pass is implemented by function ‘`final`’. As a main helper function, it uses ‘`final_scan_insn`’, which is called for each instruction of the current function, and which controls the translation to assembler code. To this end, it determines the ‘`insn_code_number`’ (see ‘instruction code’ in Appendix A) of the current instruction by trying to have it recognized using macro ‘`recog_memoized`’. If this is successful, ‘`get_insn_template`’ will be called in order to get the corresponding ‘output template’ returned as a string. For example, if ‘`insn`’ matched the ‘`addsi3`’ pattern, the output template would be ‘`add %1, %2, %0`’. Since the template string still contains magic characters, which need to be replaced by real operands, it is passed to ‘`output_asm_insn`’ along with the vector of RTL operands of the current instruction. Function ‘`output_asm_insn`’ scans the template and passes most of the characters through to ‘`asm_out_file`’ unchanged. Whenever it sees a ‘`%`⟨*digit*⟩’, however, it calls ‘`output_operand`’ to have the RTL operand ‘`operands[`⟨*digit*⟩`]`’ translated to a valid assembler operand. These are also written to the assembler output file.

The final pass ends when the templates of all instructions of the current user function have been processed. The compiler will then proceed to the next function, if any more functions are present.

2.3.8. Machine Description

As already mentioned in the overview of the compiler stages in Section 2.3.2, 'cc1' has to be made aware of the features and instructions available on the target machine by implementing a 'machine description'.

A machine description for a new target architecture '⟨*arch*⟩' consists of several files, which by convention are put under the directory 'config/⟨*arch*⟩/' in the GCC source tree. For example, the implementation of a machine description for 'C6x', which is discussed in Chapter 3, has been added to the GCC source tree under directory 'config/c6x/' for the purposes of this work. Integral parts of each machine description are '⟨*arch*⟩.md', which is a file of instruction patterns, and '⟨*arch*⟩.h', which is a C header file.

The '.md' file contains various kinds of pattern definitions, of which the most commonly used are 'instruction patterns', described in Section 2.3.9, and 'expander definitions', described in Section 2.3.11. Expander definitions tell the compiler how to generate machine-specific RTL code for certain operations (arithmetic, jumps, calls, value-moving) that must be performed. For each generated piece of RTL, a matching instruction pattern must be provided in order to make it possible for the compiler to translate it to assembler statements for the target machine. Therefore, expander definitions are used during RTL generation, described in Section 2.3.6, whereas instruction patterns are mainly used in the final pass, described in Section 2.3.7. The patterns of the '.md' file are parsed at 'build-time' by a number of modules called 'genemit.c', 'genconditions.c', 'genflags.c', etc., which the 'make' environment builds prior to any other compiler module. These tools generate new C source and header files, whose names by convention are composed of 'insn-' and the part after 'gen' of the generating tool, e.g. 'genemit.c' generates 'insn-emit.c'. The new modules will be built along with the rest of the compiler modules. This allows them to provide the information from the '.md' file in form of C data structures and C functions, which can be accessed more efficiently.

In the '.h' file, macro definitions for configuring various aspects of the compiler can be specified. GCC uses these macros, of which more than 500 exist, at places in the source code where decisions have to be made that depend on the peculiarities of the current target machine. In the following, some of these instances shall be briefly discussed.

- Types: The size of certain C types can vary on different target machines. For example, on some machines, the bit width of type '`long int`' is equal to the width of type '`int`'. On other machines, the widths differ. Macro '`INT_TYPE_SIZE`' and macro '`LONG_TYPE_SIZE`' can be used to control the sizes of these integer types.

- Registers: Each target machine can have its own number of hardware registers, which the register allocator needs to know when allocating homes for pseudo registers to live in. The main macro to mention here is '`FIRST_PSEUDO_REGISTER`'. It specifies the first register number that is a pseudo register. As the hard registers occupy a range from zero to one less than the first pseudo register, the number of hard registers is implied by this. Typically, some of the hard registers are used for fixed internal purposes (e.g. frame pointer, stack pointer, pointers to global tables) and should therefore be hidden from the register allocator. This can be achieved by providing an appropriate definition for the 'FIXED_REGISTERS' array initializer.

- Calling Conventions: In most cases, when a function is being called, a 'stack frame' has to be set up on the stack, offering space to store various kinds of information local to either the caller or the callee. No cross-platform standard exists to determine which information items should be stored in a stack frame, and in which order to store them. So, there is a great variety of options. Customarily, a stack frame contains, among others: the return address, the actual parameters passed from the caller to the callee, space for local variables and temporaries, the 'dynamic chain pointer', and the 'static chain pointer' (in the case of nested functions). These items can be stored at varying offsets, in a varying order, and they can have varying alignment requirements on different machines. While some may be allocated by the *caller*, others may be allocated by the *callee*. Parameter passing may happen in different ways: only on stack, only in registers, or both on stack and in registers. All of these aspects can be controlled by providing definitions for the target macros described in [7] under node 'Stack and Calling'. Calling conventions shall be revisited in Section 3.1.3 at the example of 'C6x'.

Some more attention will be given to target macros in Section 3.1, when discussing questions specifically related to the 'C6x' architecture.

As a supplement to '⟨*arch*⟩.md' and '⟨*arch*⟩.h', it is possible to define helper functions in '⟨*arch*⟩.c'. This is useful to keep these files small. Instead of putting complex C code into macro definitions of the '.h' file, into preparation statements of expander definitions (Section 2.3.11), or into assembler output templates of instruction patterns (see Section 2.3.9), the C code can be wrapped up in independent functions, which are put into '⟨*arch*⟩.c'. Since this module will be linked to the other GCC modules at build time, the C functions defined in it may be called from potentially anywhere in GCC and are therefore allowed to occur in macros, preparation statements, and output templates. In order to make the available C functions public to the other modules, appropriate prototypes must be put into a C header file, called '⟨*arch*⟩-protos.h', which will be automatically included via 'tm_p.h' by all compiler modules that need them (see comment at top of 'c6x-protos.h').

2.3.9. Instruction Patterns

As stated in Section 2.3.8, instruction patterns are an integral part of the machine description. They specify to the compiler which kinds of RTL expressions can directly be translated to assembler statements and how to do it. Insn patterns, as they are also called, are defined in file 'config/⟨*arch*⟩/⟨*arch*⟩.md' (where ''⟨*arch*⟩'' stands for ''c6x'' on the 'C6x' architecture), and take the following form:

```
(define_insn "addsi3"
    [(set
        ⟨template variable 0⟩
        (plus:SI
            ⟨template variable 1⟩
            ⟨template variable 2⟩))
    ]
    "⟨condition⟩"
    "add %1, %2, %0"
    [⟨attribute specifications⟩])
```

As described in [7], node ‘Patterns’, a ‘`define_insn`’ takes up to five operands, of which the last is optional:

1. Name: Genuine insn patterns just translate RTL to assembler code. In this context, the name is irrelevant and can be an empty string or a string starting with a ‘∗’, which tells the compiler to regard the pattern as nameless. Examples of such patterns in the current implementation are ‘∗`b_false`’ and ‘∗`cmp_gt`’. The part after ‘∗’ is completely ignored by the compiler and can be used as a comment for the developer of the ‘.md’ file.

   In addition to their role as genuine patterns, instruction patterns can also serve as what is called ‘expander definitions’. In this case, they need a non-‘∗’ name, which must be drawn from the set of so-called ‘standard names’. The pattern definition for ‘`addsi3`’ is an example of such a combination of two roles. For an explanation of expander definitions and standard names, please refer to Section 2.3.11.

2. RTL template: This specifies the structure of any RTX this template will be able to match. Some parts of the matched RTX are required to look exactly as specified in the template, e.g. ‘`set`’ and ‘`plus:SI`’. Other parts, which correspond to positions of template variables (also known as ‘placeholders’), are allowed to vary. However, they will only match if they satisfy a number of requirements that can be specified by means of ‘`(match_operand ...)`’ expressions at the positions of the ‘template variables’, as will be described in Section 2.3.10. For example, these requirements might allow the matching RTX to be an immediate value (‘`CONST_INT`’), but might exclude others such as register (‘`REG`’) and memory references (‘`MEM`’).

3. Condition: For nameless insn patterns, this is a string containing a boolean C expression, whose value has the last say on whether this template may match or not. When the ‘`recog_`∗’ helper functions in ‘insn-recog.c’ are generated by ‘genrecog.c’ at build-time, function ‘`write_cond`’ puts a copy of this C expression into the condition part of an ‘`if`’ statement (see case label ‘`DT_c_test`’), which guards the ‘`return`’ statement for the insn code of the current ‘`define_insn`’. Hence, even if the template structure matches, a failure of this test can prevent ‘`recog`’ from returning this insn code and make it fall back to another matching ‘`define_insn`’, if any exists.

For named insn patterns, the C expression tells whether this pattern is available for certain variations of the target machine. In addition to copying it to the appropriate ‘`recog_∗`’ function, it is processed by function ‘`gen_insn`’ in ‘genflags.c’, which generates a ‘`HAVE_∗`’ macro with the test expression as value. For example, these macros are used in function ‘`init_all_optabs`’ in ‘insn-opinit.c’, which is generated by ‘genopinit.c’ using the ‘.md’ file definitions. This shows that, as opposed to nameless patterns, tests of named patterns can get expanded in contexts outside of ‘`recog_∗`’ and can therefore not rely on any variables these functions define, most importantly the ‘`operands`’ array, which gives access to the operands already recognized.

4. Assembler output template: This is a string specifying the assembler code to output when this insn pattern matches a concrete RTL statement. The string is called ‘*template*’, because it is not a literal string, but can contain magic ‘`%`’-placeholders, which stand for operands that will later be replaced by the real operands. See [7], node ‘Output Template’ for a list of supported ‘`%`’-characters.

   Function ‘`get_insn_template`’ can be used to obtain the output template string of some instruction pattern that is given by its ‘instruction code’ (see Appendix A), which is the number of the pattern in the ‘.md’ file. ‘`get_insn_template`’ is only used in ‘`final_scan_insn`’, which is part of the final pass responsible for generating the final assembler output. The template string is passed to ‘`output_asm_insn`’, which replaces the ‘`%`’-placeholders and writes out the result.

   GCC does not access the output template string directly from the ‘⟨*arch*⟩.md’ file. Instead, it uses data structures already generated at build-time. Module ‘genoutput.c’ extracts all the strings from the ‘.md’ file and makes them available by generating C code that gets written to ‘insn-output.c’. The C code consists of variable definitions with initializers containing the template strings. Also, a definition and an initializer for variable ‘`insn_data`’ is generated. This is an array indexed by ‘instruction code’, which contains pointers to the above variables. At → ‘compile-time’, ‘`insn_data`’ will be used by function ‘`get_insn_template`’ to actually obtain the template strings.

   When written in the ‘.md’ file, output templates are double-quoted strings that exist in three different forms, which are distinguished by the first character:

   - Simple string: This may neither start with ‘`@`’ nor with ‘∗’. It contains something like ‘`add %1, %2, %0`’, which will later be output as assembler code after replacing the ‘`%`’-placeholders with real operands.
   - ‘`@`’-template: Contains several templates of the ‘simple string’ kind. Each new line corresponds to a separate ‘alternative’ (see Appendix A for a definition of ‘alternative’). Eventually, only one alternative will be chosen and the corresponding line will be output the same way as a ‘simple string’. Multiple instructions can be packed into the same line by separating them with ‘`\;`’.
   - ‘∗’-template: This has not been used in ‘c6x.md’, but shall be mentioned here for completeness. When extracted by ‘genoutput.c’ (more precisely: function ‘`process_`

`template`'), this kind of template is taken as it is and is put into the body of a new C function which is written to 'insn-output.c'. When the compiler is being built, object code for this function will be generated, so that, at compiler runtime, '`get_insn_template`' can call the function when it wants to obtain the simple string to use for generating the assembler output.

5. Attribute values: When this insn pattern matches a concrete instruction and function '`constrain_operands`' has determined the 'constraint alternative' to use (result is put into variable '`which_alternative`'), the compiler can look up the assembler output template that is needed to generate assembler code. Sometimes, however, the compiler needs to know more than that.

   For example, on a machine supporting parallel execution, a particular instruction may use certain functional units or other resources of the CPU, which should not be used at the same time by any other instructions. So, if another insn shall be scheduled to the same time slot, the compiler needs to know exactly which resources remain available after scheduling the first instruction in order not to generate invalid assembler code.

   Another example would be the length of a binary instruction word in memory. When the compiler has to decide whether it may output the short form of branch instruction for a relative jump, it needs to know an upper bound of the lengths of the instructions that are to be jumped over. Short ranges can be specified as immediate operands, whereas with long jumps, a temporary register is required.

   Meta information such as just described, cannot be deduced from the output template, but must be computed independently. In GCC, the 'attribute' mechanism is used for solving this task.

   In the practical part of this work, no attributes have been defined or used, so their syntax shall not be discussed, here. For details on how to define new types of attributes and how to set their value in insn patterns, please refer to [7], node 'Insn Attributes'.

   Similar to assembler output templates, attribute specifications are extracted from the machine description at build-time in order to generate C data structures and functions. The extraction is performed by 'genattrtab.c', whose outputs are put into file 'insn-attrtab.c'. When finished, this file will contain definitions for '`get_attr_`$*$' functions that can be used to obtain the attribute values for a RTX at compiler runtime. For example, '`get_attr_type`', can be used to compute the attribute value of the '`type`' attribute, if it has been defined in the '.md' file.

2.3.10. Template Variables

Template variables, which occur as part of 'RTL templates', described in Section 2.3.9, deserve special mention, because they are key to much of the flexibility that lies in using instruction patterns for describing machine capabilities. Template variables come in several forms, which are described in [7], node 'RTL Template'.

The most important form, which is used heavily in the file ‘`c6x.md`’, is ‘`match_operand`’. The instruction pattern definition for the standard name ‘`addsi3`’ contains an example of such an expression, which shall be quoted here:

```
(match_operand:SI
    2
    "general_operand"
    "r"
    )
```

The corresponding general form would be:

```
(match_operand:⟨Machine Mode⟩
    ⟨Operand Number⟩
    ⟨Predicate Function⟩
    ⟨Constraint List⟩
    )
```

The details of these parts are described in [7], node ‘RTL Template’, so just a short summary will be given, here:

1. Machine Mode: Describes the size of the data object that is used on the target machine. ‘`SI`’ means ‘Single Integer’ mode, which always means ‘4-byte integer’ in GCC. Other machine modes are listed at [7], node ‘Machine Modes’.

2. Operand Number: Each template variable must be assigned a number that is unique within its containing RTL template. Numbers must be chosen consecutively (checked by ‘`validate_insn_operands`’ at build-time), starting from zero. Even though the numbers are often used in ascending order within the RTL template, they may be used in any order that is desired. This may seem a redundant feature, but is important for certain expander definitions such as ‘`ble`’ in ‘`c6x.md`’. When function ‘`insn_extract`’ extracts the operands of an RTL instruction to the ‘`recog_data`’ data structure, it uses the operand number of each template variable as an index into its ‘`operand`’ array. So, for example, the operand at the position of the template variable with operand number 2 will be extracted to index 2 in the array. The same array is used when ‘`output_asm_insn`’ generates the assembler code for an RTL instruction. The operand at index 2 would therefore be output by writing ‘`%2`’ in the assembler output template.

3. Predicate Function: This is a string containing the name of a C function, which must accept an ‘`rtx`’ and an ‘`enum machine_mode`’. When a concrete RTL instruction shall be recognized, the RTX and the machine mode of each of its operands will be passed to the corresponding predicate function, which is that of the template variable corresponding to the operand’s position in the instruction. If the operand does not satisfy the requirements of the predicate function, ‘false’ will be returned and the whole instruction pattern will fail to match the instruction.

Predicate functions can therefore be used to make an instruction pattern allow only very specific kinds of RTX to occur at certain positions in an instruction. Some predefined predicate functions can be found in 'recog.c': '`general_operand`' (allows any kind of operand), '`memory_operand`' (memory references), '`register_operand`' (register references), and '`immediate_operand`' (constants).

4. Constraint List: This is a string containing a comma-separated list of 'constraints'. In most cases, a constraint is a letter that represents the type of operand, to which this template variable must be converted in order for the final pass to be able to generate valid assembler code. For instance, the constraint '`r`' in the variable with operand number 2 would tell the compiler that a register is required at index 2, which will be substituted at '`%2`' in the assembler output template.

   Constraints are different from predicate functions in that they do *not* cause any instruction pattern to fail. Instead, if a pattern matches a given RTL instruction, the constraints are looked at by the '`reload pass`' and tried to be satisfied by emitting appropriate reload instructions before and/or after the given instruction. For example, a '`(mem ...)`', which has been matched by a template variable with constraint '`r`', will be reloaded to a *register* by emitting an appropriate memory load instruction directly before the actual instruction. Instead of reading the value from memory, the instruction can now read it from the 'reload register', as it is called. This also changes the operands of the assembler instruction, which will be generated from the modified RTL instruction. Assembler instructions accepting memory operands are no longer required, but register-only instructions can now be used, which is particularly important on RISC machines like the 'C6x'.

   Each comma-delimited entry in the constraint list stands for one 'alternative'. Each template variable in an instruction pattern must support the same number of them. Alternatives are important when an instruction pattern needs to generate different assembler code depending on the kinds of operands it is matched with. For example, the '`movsi`' pattern in 'c6x.md' must distinguish register-to-memory from memory-to-register moves in order to be able to generate a store instruction in the first and a load instruction in the second case. As described in Section 2.3.9, 'assembler output templates' of the '`@`' kind can be used for specifying the output to generate for each alternative. The '`movsi`' pattern and its alternatives will be described in detail in Section 3.2.2.

2.3.11. Expander Definitions

During the RTL generation pass, the tree representation of the parsed C function needs to be expanded to semantically equivalent RTL code. As explained earlier, the compiler cannot always generate the same RTL code, but must take into account which capabilities are available on the respective target machine. This is achieved by using 'expander definitions', which have to be provided as part of the machine description for each architecture in the file 'config/⟨*arch*⟩/⟨*arch*⟩.md'.

Each expander definition needs to be given a name, which customarily is one of the more than 100 '**standard names**' that are defined in GCC. Standard names are associated with certain kinds of actions such as arithmetic computations, jumps, function calls, or move operations, which are to be emitted as RTL code.

For example, assume that the compiler has just parsed a C statement like the framed one from Section 2.3.2 and wants to expand the resulting tree representation, which is shown in Figure 3 on page 13. It has reached a point where it wants to expand the '`PLUS_EXPR`'. The compiler knows that this 'tree code' indicates an 'add' operation and therefore uses the predefined standard name '`add`⟨*mode*⟩`3`' (operation: add, machine mode: ⟨*mode*⟩, operands: two source + one target). Since normal integer variables shall be added, the ⟨*mode*⟩ to use is '`si`', which means 'single integer'. Hence, the actual name to use will be '`addsi3`'. Now, in order to expand the '`PLUS_EXPR`', the compiler will call the function '`gen_addsi3`' in 'insn-emit.c', which has been generated from the expander definition, and which will create appropriate RTL code for an 'add' operation. The RTL code can be emitted to the instruction sequence of the current function by '`emit_insn`' or one of its siblings.

Expander definitions are RTL expressions, which have expression code '`DEFINE_EXPAND`'. The '`addsi3`' example from above might have the following expander definition, for example:

```
(define_expand "addsi3"
    [(set (match_operand:SI 0 "general_operand" "")
          (plus:SI (match_operand:SI 1 "general_operand" "")
                   (match_operand:SI 2 "general_operand" "")))]
    ""
    "")
```

The general scheme followed by expander definitions is the following:

```
(define_expand "⟨standard name⟩"
    [⟨1st RTL insn pattern⟩
     ⟨2nd RTL insn pattern (optional)⟩
     ...]
    "⟨Condition⟩"
    "⟨Preparation Statements⟩")
```

The following list will explain the four operands of an expander definition. We shall use the same names as in [7], node 'Expander Definitions'.

1. Name: The name that shall be given to this definition. If this is a 'standard name', the compiler will be made aware of this pattern by adding it to the internal table for standard names, which is called '`optab_table`'. When some C statement in the user code requires a certain kind of action to be output as RTL code, the compiler will in most cases look into '`optab_table`' for the appropriate 'instruction code' to use. When trying to find instances of this in the GCC source code, please note that most of the slots in '`optab_table`' have macros that are used as aliases to access them. The definitions of these macros can be found in 'optabs.h'. Examples are: '`add_optab`', '`sub_optab`', and '`cmp_optab`'. Please refer to Section 2.3.12 for more on Optab tables.

   As already mentioned in Section 2.3.9, one instruction pattern ('`define_insn`') can be used in two roles, if it has a 'standard name'. The first one is the *nameless* role (described earlier), in which the pattern is interpreted as being genuine, and is used for RTL recognition and assembler output generation. The second role is the *named* one, in which the pattern is interpreted as an expander definition for some 'standard name'. In this role, the RTL template will not be used for RTL recognition, but rather serve as a template for RTL *generation*. Of course, the assembler output template of the '`define_insn`' is meaningless in this context and will be ignored. Conversely, no 'preparation statements' (described shortly) can be specified, since the '`define_insn`' has no operand for it. The benefit of using such a combined definition is that, where applicable, the RTL template has to be written only once for both the insn pattern and the expander definition, which makes it harder to make a mistake in the implementation. In fact, the above example of '`addsi3`' is actually implemented as a named '`define_insn`' rather than a '`define_expand`' in the current machine description. Other examples of combined definitions are '`movsi`' and '`indirect_jump`'.

2. RTL template: This is a vector of one or several RTL expressions, each typically containing some template variables. The vector serves as a template for generating concrete RTL instructions when this expander definition is called. Calling an expander definition '⟨*stdName*⟩' actually means to call its associated function '`gen_`⟨*stdName*⟩', which is generated at build-time according to the structure and contents of the RTL template. The '`gen_`⟨*stdName*⟩' function will not only reproduce the RTL template, but will also take care of replacing any template variables with actual operands that must be passed as arguments to '`gen_`⟨*stdName*⟩' (or must be generated inside of it, as explained below at 'preparation statements'). Generating the '`gen_`∗' functions is the responsibility of function '`gen_expand`' in 'genemit.c'. Function '`gen_exp`' is used as a helper function for making sure that the generated C code will produce an RTX that is structurally equivalent to the RTX described by the template, and that it will fill in the actual operands.

   For example, consider function '`gen_addsi3`' which has been generated from the RTL template definition of '`addsi3`':

```
/* ./config/c6x/c6x.md:193 */
rtx
gen_addsi3 (operand0, operand1, operand2)
     rtx operand0;
     rtx operand1;
     rtx operand2;
{
  return gen_rtx_SET (VOIDmode,
  operand0,
  gen_rtx_PLUS (SImode,
  operand1,
  operand2));
}
```

   Macro '`gen_rtx_SET`' takes two operands and generates an RTL expression of the form

'`(SET` ⟨*op0*⟩ ⟨*plus-op*⟩`)`', where '⟨*plus-op*⟩' represents the '`gen_rtx_PLUS`' expression evaluating to '`(PLUS:SI` ⟨*op1*⟩ ⟨*op2*⟩`)`'. So the returned RTX has the form:

`(`**`SET`** ⟨*op0*⟩ `(`**`PLUS`**`:SI` ⟨*op1*⟩ ⟨*op2*⟩`))`

3. Condition: This is a string containing a C expression, whose boolean value tells the compiler whether this pattern should be considered available in the current compilation. The expression typically contains flags, which can be set using appropriate command line options when invoking 'gcc'. This is mostly used when one machine description supports several variations of a target architecture, which are similar but not equivalent to each other. The flags are called 'target flags' in this case and can be defined as part of the machine description (see macro '`TARGET_SWITCHES`' and function '`set_target_switch`').

   As already mentioned with insn pattern conditions on page 24 in point 3, the condition string is processed at build-time by function '`gen_insn`' in 'genflags.c', which generates '`HAVE_`∗' macros that get written to 'insn-flags.h'. Macro '`HAVE_`⟨*name*⟩' is for expander definition '⟨*name*⟩'. The condition string of this definition is put into parentheses and written out as the replacement text of the macro. If the condition string is empty, a '`1`' is written instead.

   '`HAVE_`∗' macros, and hence also the conditions of expander definitions, get evaluated when the compiler wants to initialize the entries in the '`optab_table`' array (described in Section 2.3.12) in order to set up the 'standard name' mechanism. The initialization happens in function '`init_all_optabs`' (file 'insn-opinit.c'), which has been generated by 'genopinit.c' at build-time. When an expander definition with a non-trivial condition is to be assigned to a slot in '`optab_table`', an '`if`' statement evaluating the corresponding '`HAVE_`∗' macro will be put as a guard before the assignment.

4. Preparation statements: If not empty, this string must contain C statements, which will be inserted into the '`gen_`∗' function before the statements that were generated from the RTL template, as explained above. This way, it is possible to specify statements to be executed before the automatic RTL generation from the template happens—hence the name 'preparation statements'.

   One possible use of such statements is to create RTL expressions for additional operands, which shall occur in the RTL to be generated from the template, but which have not been passed to '`gen_`∗' as parameters and are therefore no normal operands. Such additional operands can be substituted into the RTL by writing '`(match_dup` ⟨*operand number*⟩`)`' in the RTL template and assigning to '`operands[`⟨*operand number*⟩`]`' in the preparation statements. For an example, please refer to the '`bne`' expander definition in 'c6x.md'.

   Sometimes, the RTL template shall not be used altogether, but all of the RTL shall be generated only by the preparation statements. In order to do this, the necessary RTL expressions should first be generated using a composition of '`gen_rtx_`⟨*RTX_CODE*⟩' functions. Function '`emit_insn`' should then be used to emit these expressions as insns to the local instruction sequence. A line containing '`DONE;`' should be written to finish off. Macro

'DONE' expands to a 'return' statement, whose argument has the side effect of terminating the current sequence of instructions (function 'end_sequence'). The returned value, obtained by function 'get_insns', is of type 'rtx' and points to the first insn in the sequence, which also gives access to the rest of the insns (see macro 'NEXT_INSN').

In some situations, it may make sense to let an expander definition fail, depending on the actual operands it gets passed. Note that this is not possible with the condition mechanism described above, since conditions are evaluated at a time when the compiler cannot know any concrete operands. Failure can be indicated by writing 'FAIL;'. This discards the current instruction sequence and makes the 'gen_∗' function return a null value. Note that the 'FAIL' macro can only be used with certain standard names. If used elsewhere, it might cause the compiler to abort with an internal error, since the null value is only checked for in selected places. The current implementation does not use 'FAIL'.

2.3.12. Optab Tables

At the end of Section 2.3.11 in point 1, it was mentioned that the compiler initializes an array called 'optab_table' (short for operator table) to hold information on which expander definitions to use when a certain kind of operation occurs in the user code. Actually, GCC maintains some more of these tables. Examples are 'extendtab', 'fixtab', and 'floattab', which are defined in 'optabs.c'. The reason they exist is to gain flexibility in working with 'standard actions' by introducing a level of indirection.

Entries in 'optab_table' are of type 'optab', a pointer to 'struct optab', which is defined in 'optabs.h'. Each entry comprises a number of 'standard names', which all do basically the same operation, but expect their operands in different 'machine modes'. For example, the instruction codes of 'addqi3', 'addhi3', and 'addsi3' can all be accessed by writing:

add_optab->handlers[(int) ⟨***enum machine_mode***⟩].insn_code

This allows the basic type of an operation (e.g. addition) to be handled independent of the machine mode of the operands, so that there is no need for implementing separate functions when some operation is to be performed several times on operands of varying machine mode.

Further, groups of similar operations such as 'xor', 'and', and 'or' can often be treated the same way, as far as performing a certain kind of optimization on their operands is concerned. Introducing variables like 'the_optab' (of type 'optab'), which can represent any of these operations, allows to write more general code and to reduce redundancies, because all three operations can now be handled by a single function, if so designed. For example, in order to generate RTL for a not further specified operation, a function could use

the_optab->handlers[(int) **SImode**].insn_code

to find the instruction code of the appropriate expander definition to be used for operands of machine mode 'SImode'.

Some types of operations do not fit the scheme of ‘`optab_table`’, because they require *two* rather than *one* machine mode parameters to unambiguously determine their implementing instruction code. Such operations have been split off from ‘`optab_table`’ and put into independent tables such as ‘`extendtab`’ (extending integers from one mode to another), ‘`fixtab`’, and ‘`floattab`’ (conversion from a floating point mode to a fixed point mode and vice versa).

‘`optab`’ variables such as ‘`the_optab`’ can potentially represent any operation. Sometimes, however, functions accepting ‘`optab`’ variables need some additional information on the properties of the operation described by the variable. Interesting questions might be, for example, whether an operation is commutative, whether it is unary, whether it is a comparison operation, and so on. To allow for a function to find out, ‘`struct optab`’ contains the field ‘`code`’ of type ‘`enum rtx_code`’, which holds the type of RTX whose semantics is closest to the operation this ‘optab’ represents. Which this is in detail is initialized by function ‘`init_optabs`’, which calls ‘`init_optab`’ for each optab in order to set its ‘`code`’ field. The RTX code can be passed to macro ‘`GET_RTX_CLASS`’ to obtain its class. A list of defined classes can be found at the top of file ‘rtl.def’. Among them are those for commutative, unary, and comparison operations. A reverse mapping is set up in the array ‘`code_to_optab`’.

At some point during startup of the compiler, ‘`optab_table`’ and the other tables have to be initialized. This happens in function ‘`init_all_optabs`’. The call chain for this is:

$$\text{‘lang\_dependent\_init’} \xrightarrow{\text{calls}} \text{‘init\_optabs’} \xrightarrow{\text{calls}} \text{‘init\_all\_optabs’}$$

The code of function ‘`init_all_optabs`’ is generated by ‘genopinit.c’ (at ‘build-time’), which uses an array of patterns called ‘`optabs`’. Each of these patterns is a template for a C statement assigning the ‘instruction code’ for some standard name to its slot in the appropriate operator table, e.g.:

```
‘"add_optab->handlers[$A].insn_code = CODE_FOR_$(add$P$a3$)"’
```

The patterns contain markup of the form ‘`$`⟨*char*⟩’, which is processed by function ‘`gen_insn`’ in ‘genopinit.c’. ‘`$a`’ represents the short form of a machine mode such as ‘`si`’. ‘`$A`’ represents the same machine mode, but written in its long form, which would be ‘`SImode`’ in this case. Other valid markup characters are explained in the comment at top of file ‘genopinit.c’. Module ‘genopinit.c’ works as follows: Function ‘`main`’ goes through all ‘`DEFINE_INSN`’ and ‘`DEFINE_EXPAND`’ patterns in the ‘.md’ file and calls function ‘`gen_insn`’ for each. ‘`gen_insn`’ tries to match the name of the current pattern with the ‘`$(...$)`’ part of one of the patterns in the ‘`optabs`’ array. If a matching pattern in ‘`optabs`’ is found, ‘`gen_insn`’ will instantiate it, replacing any ‘`$`⟨*char*⟩’ parts by appropriate strings and discarding ‘`$(`’ and ‘`$)`’. The resulting C statement will be added to the body of ‘`init_all_optabs`’. If the definition pattern contained a condition as described in sections 2.3.9 and 2.3.11, the C statement will be conditionalized by putting it inside a ‘`if (HAVE_`⟨*standard name*⟩`)`’ statement.

3. Implementation Details

One aim of this work was to implement basic support for the TI TMS320-C6000 architecture as a target in GCC. This chapter will discuss the efforts that have been made in this direction and their results. It shall be worked out which target macros, expander definitions, and instruction patterns are necessary for a new target and how they are currently defined.

3.1. Target Macros for C6x

Target macros are GCC's way to adapt itself to the peculiar set of features and to the parameters of a new target machine. For example, macros are used for configuring the number of available registers and the special usage of these registers (e.g. integer, floating point operations). There are macros for specifying the stack frame layout, which is important for function calls. Also, the format of the assembler output can be controlled by target macros.

GCC knows more than 500 target macros definable in 'config/c6x/c6x.h'. Explaining the semantics of all of them would be an enormous task, which is beyond the scope of this work. Fortunately, however, it turned out that not all of the 500 macros always need to be defined. The present implementation of 'C6x' defines only a few more than 130 of them and is already able to translate a reasonable subset of the C language to assembler code. Comparing this to the 'vax' target, which has less than 160 macro definitions, not many more macros should be needed to provide a fully working C compiler.

The following subsections will focus on the configuration of registers, register classes, the stack frame layout, and assembler output. The most important macros and those with rather complex implications will be discussed. As for the rest, please refer to the GCC source code in order to find out about their context of usage in the compiler. The comments in 'config/c6x/c6x.h' may also be helpful in understanding the precise semantics of each macro. Often, the comments also contain a hint as to whether providing a definition is optional or mandatory. Please note that considerable parts of 'config/c6x/c6x.h' have been copied from [7], which is another good source of information.

3.1.1. Registers

CPU registers are faster than memory in almost all cases. Therefore, in order to generate efficient code, it is desirable for the compiler to keep values of variables in registers for as long as possible, and only to put them in memory when it is absolutely unavoidable. The more CPU registers are available, the more 'pseudo registers' can be allocated to 'hard registers' by the 'register allocator' and the less will end up in memory slots on the stack.

The C64x has 64 general purpose registers: '`A0`'–'`A31`' and '`B0`'–'`B31`'. C62x and C67x both have only 32 registers: '`A0`'–'`A15`' and '`B0`'–'`B15`'. These assembler names hardly matter to GCC before it goes about outputting the final assembler code. As long as it is working on RTL code, registers are addressed just by their numbers. Macro **'`FIRST_PSEUDO_REGISTER`'**, which is currently defined to '`64`', can be used to limit the number of hard registers.

This does not automatically mean that 64 registers are available for register allocation. Typically, there are some registers on each architecture which are reserved for fixed purposes. For example, there are often two registers serving as the 'stack pointer' and the 'frame pointer'. Such registers must be excluded from register allocation, because they need to retain their values from the beginning to the end of a function and must not be clobbered by storing other values into them. Macro **'`FIXED_REGISTERS`'** can be used to achieve this. The macro is used as an initializer for array '`initial_fixed_regs`', which is indexed by register number. The register allocator will ignore any registers that correspond to non-zero entries in this array and will never allocate pseudo registers to them[3]. On 'C6x', '`A14`', '`A15`', and '`B15`' are currently marked as fixed registers. '`A14`' is used as an internal scratch register, **'`A15`'** is the **frame pointer**, and **'`B15`'** used as the **stack pointer**.

This mechanism is useful not only to mark existing CPU registers as fixed, but also to prevent RTL for *non-existing* CPU registers from ever being generated. In the case of C62x and C67x, this could be used to mark '`A16`'–'`A31`' and '`B16`'–'`B31`' as non-existing[4]. However, this cannot be achieved by means of the initializer from above, since the type of sub-architecture can only be set by a '`-m`' option and is not available before GCC runtime. The solution to this problem is the optional macro **'`CONDITIONAL_REGISTER_USAGE`'**, which should be defined to a compound statement modifying entries in the '`fixed_regs`' array (see footnote 3). A definition for this macro might look similar to the one in 'config/alpha/alpha.h'.

For the detection of clobbers, a special problem arises when a function call occurs, since the instructions in the called function could potentially clobber any register. In order to still allow for the calling function to keep some of its registers live across a function call, macro **'`CALL_USED_REGISTERS`'** can be defined to an array initializer of a form similar to '`FIXED_REGISTERS`'. The semantics of the entries is different, however: Each '`0`' represents a register that is guaranteed not to be clobbered by any function call, whereas a '`1`' indicates that the corresponding register needs to be saved in the 'caller save area' if it shall be kept live across the call. The latter case ('`1`', i.e. call-used) is taken care of automatically by function '`setup_save_areas`'. For the former case, special arrangements have to be made in the called function for storing the affected registers in the function prologue and restoring them in the epilogue, as is explained in Section 3.2.1.

3.1.2. Register Classes

Many architectures have several types of CPU registers, dedicated for example to integer, floating point, or SIMD operations. The concept of register classes allows GCC to distinguish these types. Target macros are provided which allow to define each class in terms of its associated registers and to give it a name. Also, for each register class, an individual letter can be defined as a tag and can subsequently be used in 'constraint' specifications of template variables in definitions of

[3]This is a slight simplification. Actually, '`initial_fixed_regs`' is not used directly, but is copied to '`fixed_regs`' by function '`init_reg_sets`'. Array '`fixed_regs`', in turn, is used to initialize '`fixed_reg_set`' in function '`init_reg_sets_1`', which is the one used by the register allocator for excluding registers.

[4]Has not yet been implemented.

Register class	Comment
• ‘`NO_REGS`’	A dummy class which is empty. Required by the local and global register allocator, for example.
• ‘`CONDITION_A_REGS`’	Represents the registers from file ‘A’ which can be used as condition registers on ‘C6x’.
• ‘`CONDITION_B_REGS`’	Like ‘`CONDITION_A_REGS`’, but for register file ‘B’.
• ‘`A_REGS`’	File ‘A’ registers.
• ‘`B_REGS`’	File ‘B’ registers.
• ‘`ALL_REGS`’ aliased by ‘`GENERAL_REGS`’	All registers, i.e. the union of ‘`A_REGS`’ and ‘`B_REGS`’.

Table 3: Register Classes on ‘C6x’

instruction patterns. Register classes can contain smaller register classes, so that a hierarchy from special (small classes) to general (large classes) registers can be defined. A good introduction to register classes can be found in [7] at node ‘Register Classes’.

The ‘C6x’ architecture has two register files called ‘A’ and ‘B’, which are connected to the ‘A’ and ‘B’ side of functional units, respectively. Instructions to be executed on one of the ‘A’ units can use registers from the ‘B’ file only via the cross path ‘1X’. If there is a second instruction to be executed in parallel, it will only be able to use registers from file ‘A’, since the cross path is already occupied. Therefore, when support for the parallel execution features of ‘C6x’ shall be implemented in the backend, it will be necessary to distinguish ‘A’ and ‘B’ registers. With the current implementation, since only one instruction per CPU cycle is executed, the cross path is always available, and the machine description could possibly manage without separating the registers. However, in order to be prepared for future developments, the separation has been implemented anyway, and is described in the following.

Firstly, an **‘`enum reg_class`’** containing all register classes had to be defined. All classes currently existing on ‘C6x’ are listed in Table 3. Macro **‘`REG_CLASS_NAMES`’** is an initializer for ‘`char *reg_class_names[]`’, which should be defined to contain a list of string literals corresponding to the names of the classes. Next, the contents of each class had to be defined by macro **‘`REG_CLASS_CONTENTS`’**, which is used as initializer of array ‘`int_reg_class_contents[`⟨*num_reg_classes*⟩`][`⟨*num_reg_ints*⟩`]`’. The entries in this variable are integers, which are interpreted as bit masks of width 32. ‘C6x’ has 64 registers, hence two masks are needed per class, where ‘`...[`⟨*class*⟩`][0]`’ represents registers 0–31 and ‘`...[`⟨*class*⟩`][1]`’ registers 32–63. For example, to indicate register 34 being a member of register class 3, bit 2 in ‘`int_reg_class_contents[3][1]`’ should be set. A more general explanation of the format of this initializer can be found in [7] at node ‘Register Classes’. The mapping just described translates from class number to the contained register numbers. GCC also requires the reverse mapping to be defined using macro **‘`REGNO_REG_CLASS(`⟨*regno*⟩`)`’**. The macro should be a C expression, which compares ‘⟨*regno*⟩’ to a number of ranges corresponding to register classes, and which evaluates to the number of the smallest class containing ‘⟨*regno*⟩’. The current definition looks as follows:

```
#define REGNO_REG_CLASS(REGNO) \
  (   (REGNO) >=  1        && (REGNO) <=  2        ? CONDITION_A_REGS   \[5]
   : (REGNO) >= 32         && (REGNO) <= 34        ? CONDITION_B_REGS   \
   : (REGNO) >= 0          && (REGNO) <= 31        ? A_REGS             \
   : (REGNO) >= 32         && (REGNO) <= 63        ? B_REGS             \
   : NO_REGS)
```

Some more macros could be mentioned here, which define characteristics of register classes, but we shall confine ourselves to one more macro, which is called **'`REG_CLASS_FROM_LETTER`'** and is used, for example, by function '`constrain_operands`' to translate the letters used in constraint specifications of template variables to the corresponding register classes. On 'C6x', it is currently defined as follows:

```
#define REG_CLASS_FROM_LETTER(C) \
  (   (C) == 'a' ? A_REGS                                               \
   : (C) == 'b' ? B_REGS                                                \
   : (C) == 'A' ? CONDITION_A_REGS                                      \
   : (C) == 'B' ? CONDITION_B_REGS                                      \
   : NO_REGS)
```

3.1.3. Stack Frame Layout

This section describes which target macros are available for controlling the layout of stack frames in GCC. Once finished with reading, you might want to refer to Table 4 on page 43 for a summary.

The stack plays a vital role in implementing the concept of functions in C and most other programming languages. Typically, 'stack frames', also referred to as 'activation records' (see Appendix A), are used as temporary memory that is valid for as long as a function or any functions called from within it are active.

A '**stack frame**' is a contiguous piece of stack memory that gets allocated (i.e. pushed) each time a function is called and is destroyed (i.e. popped) when the function returns. Hence, the stack frame of the most recently called function, which is at the bottom (a leaf) of the calling hierarchy, is topmost on the stack. Functions on the way up the calling hierarchy have frames lying deeper in the stack. Customarily, each stack frame contains a 'dynamic chain pointer' pointing to the frame of the caller of the current function. It is called *chain*, because dereferencing the pointer once more would yield a pointer to the frame of the caller's caller, which could be dereferenced further until the frame of function '`main`' was reached.

Since stack frames live exactly as long as their corresponding function is active, they are well-suited to store any data to be allocated at the start or in the course of a function and to be automatically destroyed on return. Typical examples of such data are local variables (also called 'automatic variables'), function arguments, the return address, temporary values from expression evaluation, areas for registers to be saved on entry and restored on exit, and '`alloca`'-allocated dynamic memory areas. The stack frame layout currently implemented for the C6x target is depicted in Figure 4 on page 39.

[5]Note that '`A0`' is excluded here, because it is not allowed as a condition register on 'C62x' and 'C67x'. In order to enable it on 'C64x', this defi nition should be changed so that it depends on 'target flags' (see Appendix A).

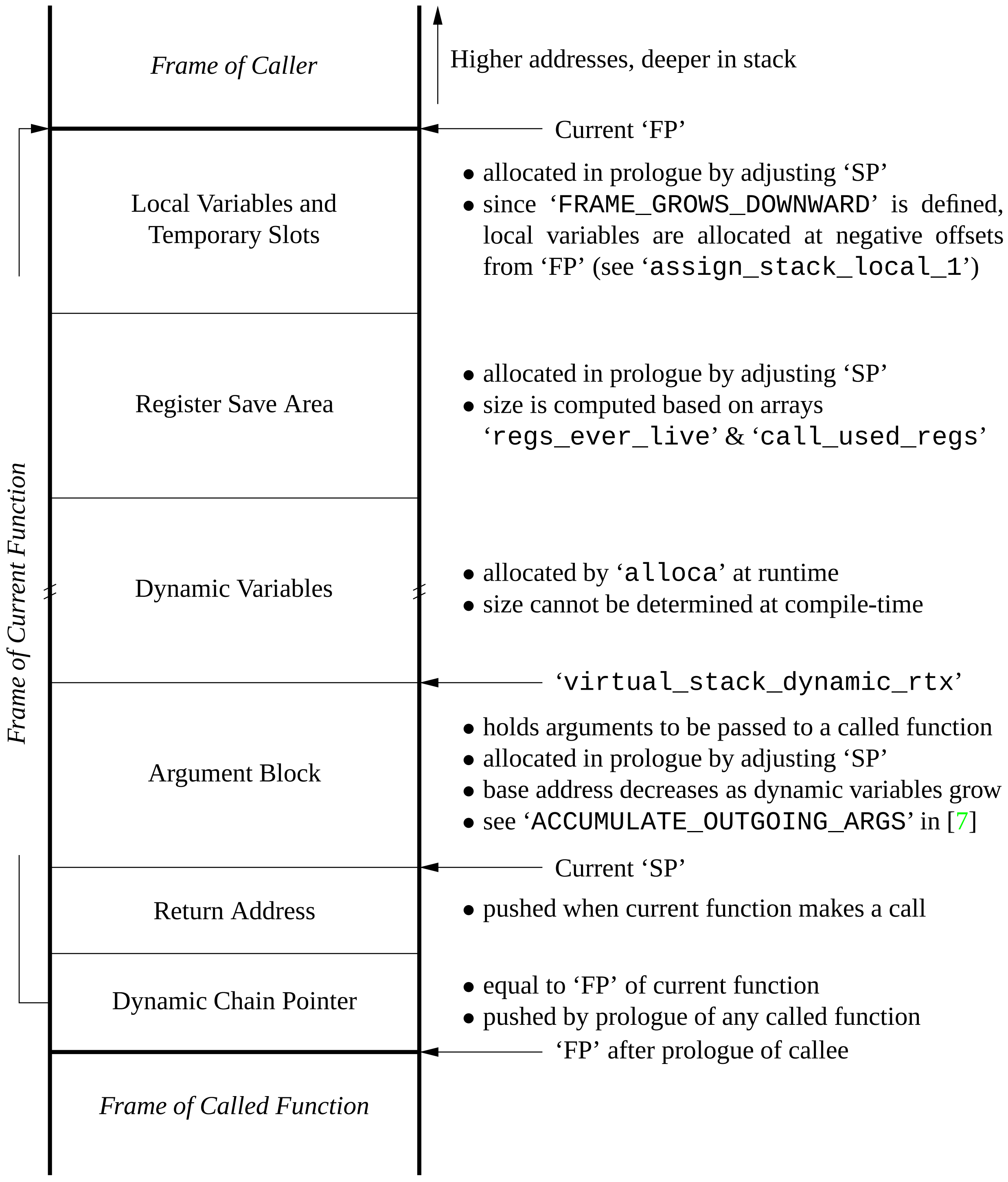
Frame of Caller
Higher addresses, deeper in stack
Current 'FP'
Local Variables and Temporary Slots
• allocated in prologue by adjusting 'SP'
• since 'FRAME_GROWS_DOWNWARD' is defined, local variables are allocated at negative offsets from 'FP' (see 'assign_stack_local_1')
Register Save Area
• allocated in prologue by adjusting 'SP'
• size is computed based on arrays 'regs_ever_live' & 'call_used_regs'
Dynamic Variables
• allocated by 'alloca' at runtime
• size cannot be determined at compile-time
'virtual_stack_dynamic_rtx'
Argument Block
• holds arguments to be passed to a called function
• allocated in prologue by adjusting 'SP'
• base address decreases as dynamic variables grow
• see 'ACCUMULATE_OUTGOING_ARGS' in [7]
Current 'SP'
Return Address
• pushed when current function makes a call
Dynamic Chain Pointer
• equal to 'FP' of current function
• pushed by prologue of any called function
'FP' after prologue of callee
Frame of Called Function
Frame of Current Function

Figure 4: Stack Frame Layout of the 'C6x' target

You can see that '**SP**' points to the base address of the argument block throughout the currently active function. To my knowledge, the only exception to this occurs when a function call pushes the return address onto the stack and the prologue of the called function sets up the new stack frame. On 'C6x', function '`c6x_expand_call`' is responsible for expanding calls (see expander definition for the '`call`' standard name). This function emits the RTX for a call, followed by a generated label '`ret_label`'. The label is important for obtaining a reference to the address after the branch instruction, which will be generated from the '`call`'-RTX by '`*call_value_internal`'. This reference represents the address to which the called function should return on exit and is therefore pushed onto the stack by calling '`c6x_emit_push`'. This is shown as the 'Return Address' box in Figure 4.

Where precisely to draw the borderline between the frame of the current function and the called function is certainly disputable. One might take the position that the stack frame should end at the point where the branch instruction jumping to the called function is executed. Another way to look at it, however, is to say that the name 'frame pointer' suggests that '**FP**' is a *pointer* to the base address of the stack *frame*. The author decided to take the latter view and to draw the line there, ignoring the fact that the 'dynamic chain pointer' should actually rather not belong to the current function, because it is the *called* function that pushes it in its prologue.

The main reason why a frame pointer is required at all in addition to the stack pointer is the fact that the size of the stack frame cannot generally be computed at compile-time. This is because the 'dynamic variables', shown in Figure 4, can be allocated in arbitrary quantities using function '`alloca`' at runtime. Using 'SP' as the base register for addressing the local variables, for example, is therefore impossible. Instead, another register on the other side of the dynamic variable block is needed, which is the frame pointer.

Local variables and temporary slots are allocated by function '`assign_stack_temp_for_type`', which either reuses an existing but abandoned stack slot or allocates a new one by calling '`assign_stack_local`', which is a wrapper for '`assign_stack_local_1`'. When allocating a new stack slot, '`cfun->x_frame_offset`' is increased or decreased by the size of the new slot. In the current implementation, it is decreased, because macro '`FRAME_GROWS_DOWNWARD`' is defined. The initial value of field '`x_frame_offset`' is zero (see init of '`cfun`' in '`prepare_function_start`'), so that the slots get allocated at negative offsets. As the base register, the frame pointer is used, which is represented by '`frame_pointer_rtx`' (initialized by function '`init_emit_once`')[6]. As an example, let us assume that the first slot allocated in the current function was of type '`int`' and therefore had a size of 4 byte. The RTL generated for referencing it in memory would be '`(mem:SI (plus:SI (reg:SI 65) (const_int -4)))`', where 65 would be the register number of '`virtual_stack_vars_rtx`'. In case there were any other integer variables, they would be placed at offsets -8, -12, and so on.

A few remarks on the **'dynamic variable' block** remain to be made: This block is special in that it can grow dynamically and multiple times in the course of a function. Each

[6]Instead of '`frame_pointer_rtx`', virtual register '`virtual_stack_vars_rtx`' may also be used, but later gets replaced by '`frame_pointer_rtx`' in function '`instantiate_virtual_regs_1`' (for virtual registers, see page 11).

time the user code contains a call to '`alloca`', the block is increased by an amount determined by the C expression, which is provided as size argument to this function. In GCC, '`alloca`' is a macro defined to function '`__builtin_alloca`', which is expanded by '`expand_expr`'$\xrightarrow{\text{calls}}$'`expand_builtin`'$\xrightarrow{\text{calls}}$'`expand_builtin_alloca`'. This function emits code for evaluating the size argument and passes an RTX holding the result to function '`allocate_dynamic_stack_space`', which does two things: Firstly, it calls '`anti_adjust_stack`' to push the requested space onto the stack. Secondly, it emits a move instruction saving the new value of '`virtual_stack_dynamic_rtx`', which is RTX for the base address of the newly allocated space. The value is saved to the location represented by RTX '`target`', which is the return value from '`alloca`' as the user program will see it. '`virtual_stack_dynamic_rtx`' is a virtual register that later gets instantiated to '`stack_pointer_rtx`' plus '`STACK_DYNAMIC_OFFSET`'. The latter is a macro that evaluates to '`current_function_outgoing_args_size`'[7], whose value determines the size of memory the function prologue (see '`c6x_expand_prologue`') allocates for the 'argument block' (shown in Figure 4).

The purpose of the '**argument block**' is to serve as a preallocated buffer where argument values to be passed to a function are stored into, immediately before the function call. In order to be able to address the dynamic variables by means of 'SP' as base register, '`STACK_DYNAMIC_OFFSET`' must be fixed and determinable at compile-time. Hence, the argument block cannot grow or shrink dynamically, but must be allocated of correct size right from the start. It must therefore be large enough to accommodate for the maximum space needed in any call made in the current function. For example, assume that the only functions called by the current function were '`void f1 (int)`' and '`void f2 (int, int)`', then '`current_function_outgoing_args_size`' would amount to 8, because only 4 bytes would be required for '`f1`', but 8 bytes would be required for passing the two integers to '`f2`'.

Since the argument block is used by both the current function (here denoted as 'caller') and any called function (here denoted as 'callee'), they both need access to it. This is complicated by the fact that there is a frame switch between the time when the caller writes the arguments and the time when the callee reads them back. This switch changes the values of 'SP' and 'FP', so that neither can be used as a common base register for addressing the argument block. For this reason, GCC makes the caller and the callee use two separate mechanisms:

1. Caller: When function '`expand_call`' expands a function call, it uses function '`compute_argument_addresses`' to have a slot of the argument block assigned to each entry of array '`args`', which is the list of actual arguments as extracted from '`actparms`' by function '`initialize_argument_information`'. To represent the slots in RTL, '`mem`' expressions are generated using the base register given by '`argblock`', which is an RTX for the base address of the argument block and is initialized to '`virtual_outgoing_args_rtx`' by function '`expand_call`'. As can

[7] Please refer to the definition of '`STACK_DYNAMIC_OFFSET`' in 'function.c': Macro '`REG_PARM_STACK_SPACE`' is undefined on 'C6x', so the '`#else`' branch is used. Since '`ACCUMULATE_OUTGOING_ARGS`' is defined to '`1`' and '`STACK_POINTER_OFFSET`' is defined to '`0`' in the current implementation, the expression evaluates to '`current_function_outgoing_args_size`'.

be seen in function '`instantiate_new_reg`', this virtual register will later be instantiated to the stack pointer plus '`STACK_POINTER_OFFSET`', which is defined to '`0`' on 'C6x'. Hence, we can note that argument addressing on the side of the caller is performed using 'SP' as base register and requires no offset other than the slot offset.

2. Callee: Function '`assign_parms`', which is called by '`expand_function_start`' when the RTL code for a function shall be generated, is responsible for assigning RTX to the declaration node of each function parameter using the access macro '`DECL_INCOMING_RTL`'. For example, in function '`int add (int a, int b)`', the following RTL expressions would be assigned in the current implementation:

   - Parameter '`a`': '`(mem:SI (reg:SI 64))`'
   - Parameter '`b`': '`(mem:SI (plus:SI (reg:SI 64) (const_int 4)))`'

   Register 64 is the same as virtual register '`virtual_incoming_args_rtx`', which will later be instantiated to '`arg_pointer_rtx`' (argument pointer) plus '`FIRST_PARM_OFFSET`', according to function '`instantiate_new_reg`'.

   As for '`arg_pointer_rtx`', GCC allows to define '`ARG_POINTER_REGNUM`' equal to '`FRAME_POINTER_REGNUM`', which causes '`arg_pointer_rtx`' to become equal to '`frame_pointer_rtx`', so that the role of the argument pointer is completely filled by the frame pointer. The current 'C6x' implementation makes use of this method.

   As for '`FIRST_PARM_OFFSET`', it must equal the difference '*first argument address*' − '*frame pointer*', where the latter means the value of the frame pointer after the prologue of the callee has been executed. On 'C6x', macro '`ARGS_GROW_DOWNWARD`' was left undefined so that '*first argument address*' refers to the lowest address of the 'argument block'. This made it easier to find a valid definition for '`FIRST_PARM_OFFSET`'. As illustrated in Figure 4, only the 'dynamic chain pointer' and the 'return address' are currently stored between the callee's 'FP' and the 'argument block'. In order for '`FIRST_PARM_OFFSET`' to always skip exactly these two words on stack, a definition expanding to '`2 * UNITS_PER_WORD`' was chosen, which are effectively 8 bytes.

The final part of the stack frame not yet described is the '**register save area**'. It is used for saving registers that are not in '`CALL_USED_REGISTERS`' and can therefore be expected by a caller not to be modified by the current function (or any other function). Of course, the function prologue can omit any registers from saving, for which no chance of modification in the current function exists. Such a register might be found in a function that makes no calls itself and also does not otherwise write to that register. In order to facilitate the search for the correct registers to save, the compiler automatically sets up the array '`regs_ever_live`', which is indexed by hard register numbers. It contains a non-zero entry for each register that might be modified somewhere in the function. Both '`regs_ever_live`' and '`call_used_regs`' were used for implementing the function '`c6x_expand_prologue`', which is responsible for emitting RTL for function prologues on 'C6x'.

Macro	Comment
`STACK_GROWS_DOWNWARD`	If defined, GCC decreases ‘SP’ when pushing values onto the stack. Otherwise, ‘SP’ is increased. See function ‘`anti_adjust_stack`’, for example.
`FRAME_GROWS_DOWNWARD`	If defined, ‘FP’ + ‘`STARTING_FRAME_OFFSET`’ must point to the first free byte above the block of ‘local variables and temporary slots’. ‘FP’ is expected to be initialized accordingly in the function prologue. The compiler will allocate slots at increasingly negative offsets. If the first slot takes 4 byte, the first offset will be -4. This is what ‘C6x’ does. If the macro is undefined, the above sum must point to the lowest address of the block. Offsets will start at zero and will increase as required by the slot sizes. See function ‘`assign_stack_local_1`’.
`ARGS_GROW_DOWNWARD`	Similar to ‘`FRAME_GROWS_DOWNWARD`’, but with ‘AP’ + ‘`FIRST_PARM_OFFSET`’ pointing to the ‘argument block’ which has been passed to this function.
`STARTING_FRAME_OFFSET`	Offset to add to ‘FP’. See ‘`FRAME_GROWS_DOWNWARD`’.
`STACK_POINTER_OFFSET`	Amount by which ‘`virtual_outgoing_args_rtx`’ is offset from the stack pointer. This is currently defined to ‘`0`’.
`FIRST_PARM_OFFSET`	Offset to add to ‘AP’. See ‘`ARGS_GROW_DOWNWARD`’.
`STACK_DYNAMIC_OFFSET`	Amount by which ‘`virtual_stack_dynamic_rtx`’ is offset from ‘SP’. A working default definition is provided in ‘function.c’, which evaluates to the size of the ‘argument block’.
`FRAME_POINTER_REQUIRED`	If non-zero, forces the compiler to use a frame pointer in each function. In this case, ‘`frame_pointer_needed`’ will always be non-zero and the compiler will make no attempts to eliminate ‘FP’ and to use ‘SP’ instead. The macro is currently non-zero on ‘C6x’ in order to have a constant stack frame layout, which is easier to set up. See function ‘`init_elim_table`’.

(*continued on next page*)

Table 4: Target Macros controlling the Stack Frame Layout of ‘C6x’

(*continued from last page*)

Macro	**Comment**
STACK_POINTER_REGNUM	The number of the register to use as 'SP'. This is currently defined so that it evaluates to 32+15 (equivalent to 'B15'), which is in accordance with [15], section 'Function Structure and Calling Conventions' on page '8-18'.
FRAME_POINTER_REGNUM	The number of the register to use as 'FP'. This is currently defined so that it evaluates to 15 (equivalent to 'A15') in accordance with [15].
ARG_POINTER_REGNUM	The number of the register to use as 'AP'. If equal to 'FRAME_POINTER_REGNUM', no separate argument pointer will be used. Instead, 'FP' + 'FIRST_PARM_OFFSET' must point to the 'argument block' and will be used. This is done on 'C6x'.
ACCUMULATE_OUTGOING_ARGS	Must be defined to a C expression. If nonzero, the maximum amount of space required for outgoing arguments (by any function calls made in the current function) will be computed and placed into the variable 'current_function_outgoing_args_size'. The compiler will not push any space onto the stack for each call, but expects that sufficient space for the 'argument block' has already been allocated by the function prologue. On 'C6x', this macro is defined to '1'.
OUTGOING_REG_PARM_STACK_SPACE	If defined, it is the responsibility of the caller prologue to allocate not only space for any outgoing arguments passed on stack, but also for those passed in *registers*. Also, the size of arguments passed in registers will be added to 'current_function_outgoing_args_size', which already holds the size of the arguments passed on stack. Currently, this macro has no effect, since no arguments are passed in registers.

Table 4: Target Macros controlling the Stack Frame Layout of 'C6x'

3.1.4. Assembler Output

Most of the specifications the compiler needs for the translation of RTL to assembler code must be given in the 'assembler output templates', which are part of the instruction patterns described in Section 2.3.9. However, as already stated there, RTL templates often contain template variables, which get instantiated to concrete RTL operands as soon as their instruction pattern matches with a concrete RTL instruction. Later, in function ‘`output_asm_insn`’, these operands can be used for replacing the ‘`%`’-placeholders in the assembler output template by concrete assembler operands, which can be register references, memory references, or immediate values, for example. The assembler syntax to use for these operands, however, cannot be specified within the assembler output template, but must be provided by defining the two target macros ‘`PRINT_OPERAND`’ and ‘`PRINT_OPERAND_ADDRESS`’.

‘`PRINT_OPERAND`’ is the more important one of the two. The only location it is called from is function ‘`output_operand`’, which in turn is only called from ‘`output_asm_insn`’ in ‘final.c’. ‘`PRINT_OPERAND`’ takes three arguments, of which the first is the assembler output file (type ‘`FILE *`’), the second is the RTL operand to be printed (type ‘`rtx`’), and the third is the ‘⟨*char*⟩’ of a ‘`%`⟨*char*⟩⟨*digit*⟩’-placeholder (type ‘`int`’). On ‘C6x’, the macro does not perform any output itself, but simply passes all of its arguments to ‘`c6x_print_operand`’, which is implemented in ‘config/c6x/c6x.c’. If no ‘⟨*char*⟩’ is specified, zero will be passed in the ‘`code`’ argument. In fact, this is the only case currently supported in ‘`c6x_print_operand`’. Other cases need not be supported, because ‘c6x.md’ currently contains only placeholders with ‘⟨*digit*⟩’, but none with an additional ‘⟨*char*⟩’. The kinds of RTL operands that are mandatory for ‘`c6x_print_operand`’ to support were found partly by looking at other machine descriptions, partly by using a debugger to get examples of RTL that is passed to this function. It turned out that three basic cases are relevant:

1. Register References: These are ‘`REG`’ expressions representing hard registers (pseudos have been replaced earlier in function ‘`alter_reg`’). Array ‘`reg_names`’, which has been initialized to the strings in macro ‘`REGISTER_NAMES`’, can be used to convert the numbers to assembler register names.

2. Memory References: These are ‘`MEM`’ expressions, of which the address must be output in assembler syntax. This task is delegated to ‘`output_address`’, which passes it on to macro ‘`PRINT_OPERAND_ADDRESS`’, which will be explained shortly.

3. Immediate Values: These can be code labels or constant literals, which are represented by expressions of code ‘`SYMBOL_REF`’, ‘`LABEL_REF`’, or ‘`CONST_INT`’, for example. Function ‘`output_addr_const`’ already implements default behaviour for outputting these expressions, so that they are handled by simply passing them on to this function.

As already mentioned, the second macro needed for outputting RTL operands in assembler syntax is called ‘`PRINT_OPERAND_ADDRESS`’. It is specialized in handling the output of address operands such as those occurring in ‘`MEM`’ expressions. On ‘C6x’, this macro just calls function ‘`c6x_print_operand_address`’, which produces the actual output. Whether an address is valid on a target machine is determined by target macro ‘`GO_IF_LEGITIMATE_ADDRESS`’, which the compiler makes use of in functions ‘`memory_address`’ and ‘`memory_address_p`’, for example. This macro is currently implemented as a wrapper for function ‘`c6x_legitimate_address_p`’. Whenever an address operand is deemed valid by ‘c6x_legitimate_address_p’, there must be a case in function ‘c6x_print_operand_address’ generating the corresponding assembler output, because otherwise the compiler will abort. Both functions currently support ‘REG’ and ‘PLUS’ expressions, where the latter must be a sum of a ‘REG’ and a ‘CONST_INT’ such as generated when allocating stack slots with offsets (function ‘assign_stack_local_1’) or instantiating virtual registers (function ‘instantiate_virtual_regs_1’). Generating output for ‘REG’ addresses is as easy as outputting the name of the register prepended by a ‘$*$’. ‘PLUS’ addresses made up of a ‘REG’ and a ‘CONST_INT’ can be output using addressing with offsets. In load and store operations, ‘C6x’ allows to use addresses with an unsigned constant offset of width 5 bits. An example of this would be ‘*–A0(12)’, which would refer to a memory location 12 bytes below that pointed to by register ‘A0’. This kind of addressing can help in achieving a higher runtime performance of the generated code, since otherwise an extra instruction would have had to be emitted for first computing the sum and then using simple addressing. However, care must be taken for the immediate offset not to exceed the allowed width of 5 bit. Addresses with larger ‘CONST_INT’ values are therefore declared invalid by ‘c6x_legitimate_address_p’, which forces the compiler to use simple addressing.

3.2. Expander Definitions and Insn Patterns for C6x

Currently, the ‘C6x’ machine description provides definitions for 26 ‘standard names’ (expander definitions and instruction patterns). This is far less than the more than 100 standard names supported by GCC. It is sufficient, however, for being able to translate simple C programs such as ‘t001.c’…‘t011.c’, which have been written as testcases for the current implementation. In the following, we shall discuss some of the implemented patterns, which will illustrate how expander definitions and instruction patterns may look like in practice. Also, some comments shall be made on which standard names are required for a working C compiler.

3.2.1. Function Prologue and Epilogue

The ‘prologue’ is code required on entry to a function for setting up the stack frame and saving any registers to be restored on exit. Analogously, the ‘epilogue’ of a function is code at the end which is needed for restoring ‘FP’, ‘SP’, and the saved registers to their old values and for performing the return from the function.

The requirements when implementing the function prologue and epilogue are described in [7] at node ‘Function Entry’. This text has been a helpful reference even though the current implementation does not use the macros ‘TARGET_ASM_FUNCTION_PROLOGUE’ and ‘TARGET_ASM_FUNCTION_EPILOGUE’ described there, but provides expander definitions for the standard names ‘prologue’ and ‘epilogue’, instead. In principle, these patterns follow the same rules as the macros, but emit the prologue/epilogue as RTL rather than assembler code. According to [7] at node ‘standard names’, this allows to extend ‘instruction scheduling’ to the prologue and epilogue, which would not have been possible with assembler code. In the current ‘C6x’ implementation, no provisions have been made to allow for instruction scheduling, so the decision in favour of RTL code was mainly made in view of possible future developments in this direction.

We shall briefly discuss the current definition provided for the ‘`prologue`’ pattern, which can also serve as an example for the ‘`epilogue`’ pattern:

```
(define_expand "prologue"
  [(const_int 1)]
  ""
{
  c6x_expand_prologue ();
  DONE;
})
```

The first of the preparation statements is a call to ‘`c6x_expand_prologue`’, a function defined in ‘c6x.c’, which generates and emits the RTL instructions of the prologue. It is followed by a use of macro ‘`DONE`’, which finishes the sequence of instructions and makes it the return value of the ‘`gen_prologue`’ function. ‘`DONE`’ also prevents the ‘`(const_int 1)`’ in the RTL template from ever being emitted.

When looking at the definition of function ‘`c6x_expand_prologue`’, one finds a sequence of actions, which is listed below. Please compare this with the stack frame layout shown in Figure 4 on page 39. The actions are as follows:

1. Push ‘FP’ onto the stack. This corresponds to assigning to the ‘dynamic chain pointer’ slot, which would have to be shown at the bottom of the ‘*Frame of Caller*’ block in Figure 4.

2. Save ‘SP’ to ‘FP’, so that it can be restored to its current value on function exit. ‘FP’ will keep its value throughout the current function. It now points to the upper limit of the stack frame to be set up for this function.

3. Decrease ‘SP’ by the size required for the block of ‘local variables and temporary slots’. This can be regarded as “allocating” this block, since subsequent push operations will leave the area untouched.

4. Find any hard registers which need to be saved, and call ‘`c6x_emit_push`’ to push them onto the stack. Currently, ‘`c6x_emit_push`’ emits an instruction for decreasing ‘SP’ by the size of the value to be pushed and then another for actually moving the value to the new address of ‘SP’.

5. Decrease ‘SP’ by the size required for the ‘argument block’.

The ‘`epilogue`’ pattern is similar to ‘`prologue`’, but calls ‘`c6x_expand_epilogue`’, which emits instructions for cleaning up the stack frame. For details, please refer to the comments in the body of this function in ‘c6x.c’.

3.2.2. Move operations

As also stated in [7] at node ‘Standard Names’, implementing the ‘movM’ patterns is mandatory. The compiler uses them whenever it needs to emit RTL for copying a value between two registers, from memory to a register, from a register to memory, or for loading a register with an immediate value. The ‘M’ in ‘movM’ stands for the machine mode this pattern supports. Currently, the pattern for machine mode ‘si’ has been implemented as follows:

```
(define_insn "movsi"
  [(set (match_operand:SI 0 "general_operand" "=r,r,m,r,r")
        (match_operand:SI 1 "general_operand" "r,m,r,M,i"))]
  ""
  "@
   mv %1, %0
   ldw %1, %0\;nop 4
   stw %1, %0
   mvkl %1, %0
   mvkl %1, %0\;mvkh %1, %0"
  [])
```

Obviously, this pattern is an ‘instruction pattern’, since it is defined using ‘define_insn’ (see Section 2.3.9). However, it is also an ‘expander definition’, because its name is not prepended by a ‘∗’. The ‘si’ in its name causes the pattern to be assigned to the ‘SImode’ slot of ‘mov_optab’ (see Section 2.3.12). Once there, it can be used from many places within the compiler. For example, it is used by function ‘emit_move_insn_1’.

Both template variables in the RTL template use ‘general_operand’ as their predicate function. This causes the pattern to match operands of almost any kind. For example, it would match an RTL expression of the form

```
(set (mem:SI (reg:SI 1)) (mem:SI (reg:SI 2)))
```

representing a move from memory at address ‘A2’ to memory at address ‘A1’. As a RISC architecture, ‘C6x’ does not offer direct support for such move operations. Interestingly, the above RTL instruction is valid anyway, because it will be processed and modified by the ‘reload pass’ according to the constraint specifications in the ‘movsi’ pattern. As can be seen from the definition above, support for five ‘alternatives’ has currently been implemented:

	Number of Alternative				
	0	**1**	**2**	**3**	**4**
Operand 0	r	r	m	r	r
Operand 1	r	m	r	M	i

Constraint ‘r’ represents a hard register of class ‘GENERAL_REGS’, which is required for this operand. If alternative 0 is chosen, the reload pass will ensure that only *register* operands (of the proper class) will be passed to the corresponding assembler output template ‘mv %1, %0’. In case one of the operands was a ‘mem’ rather than a ‘reg’, the reload pass would automatically emit a reload instruction for the respective operand. For operand 0, which is marked as an *output* operand by the '`=`' at the beginning of its constraints, there would have to be an 'output reload'. For operand 1, which is implicitly an *input* operand, an 'input reload' would be in order.

Instructions requiring *input* operands to be reloaded from a '`mem`' to a '`reg`' want to find the correct value from memory in the 'reload register' when they are executed. Therefore, the reload pass must emit the reload instruction *before* the actual instruction. For *output* operands the situation is different. The instruction first wants to write to the 'reload register' and then expects it to be written to memory. So the reload instruction must be emitted *after* the actual instruction.

When a pattern offers multiple alternatives, the reload pass will first try to find one in which all constraints are satisfied by the current instruction. Reload instructions will only be generated if no match is found. For example, assume that the current instruction is:

```
(set (mem:SI (reg:SI 1)) (reg:SI 2))
```

Operand 0 is a '`mem`' satisfying the '`m`' constraint. Operand 1 satisfies the '`r`' constraint. Hence, the '`reload`' pass will find that alternative 2 matches best (see function '`find_reloads`'). Because it even matches *completely*, the reload pass will return without emitting any reload instructions. Later, in the '`final`' pass, '`constrain_operands_cached`' will expect at least one alternative to match completely. It will again find alternative 2 and indicate this by setting '`which_alternative`' to 2, which will cause '`get_insn_template`' to select the third line (the line at index 2) in the multiple-alternative assembler output template of the '`movsi`' pattern. The third line contains the output string '`stw %1, %0`', which will subsequently be instantiated by function '`output_asm_insn`'. In order to generate assembler code for the operands, this function will pass

Operand 0: `(mem:SI (reg:SI 1))` and
Operand 1: `(reg:SI 2)`

to function '`output_operand`' and will eventually output '`stw A2, *A1`'.

This example has shown that it is possible to select different assembler output strings depending on the types of the operands occurring in a particular RTL instruction. The other alternatives that were implemented support some more cases, which we shall discuss now:

- Alternative 0: Can move from a register to another register using the **'`mv`'** instruction.

- Alternative 1: Similar to alternative 2 (discussed above), but for load rather than store operations. It uses the **'`ldw`'** instruction. The '`nop 4`' is a separate instruction that gets output directly afterwards. Its purpose is to make sure that the target register of the '`ldw`' instruction will already have been updated at the time the next instruction is executed. Without this artificial delay, the 4 delay slots of the '`ldw`' would cause the following instruction to find the old value in the register, which would lead to incorrect behaviour.

- Alternative 2: Uses the **'`stw`'** instruction as already discussed in the last paragraph.

- Alternative 3: This allows to move a small immediate value to a register. To this end, the '`M`' constraint has been introduced. It represents what [14] refers to as 'scst16', which is a signed 16 bit constant ranging from $[-8000]_{16}$ to $[7FFF]_{16}$. '`M`' is one of the machine-dependent operand constraint letters, which can be defined by target macro '`CONST_OK_FOR_LETTER_P`'. The special constraint letter was introduced because 'C6x' allows to load immediate values in the 'scst16' range with just a single '**`mvkl`**' instruction. In theory, all immediates could be handled by alternative 4, which is meant for values outside this range. But that would always require an additional '`mvkh`' instruction. So alternative 3 is a special case, saving us one instruction. In order to have a chance to be used, it must come before alternative 4 in the list of constraints.

- Alternative 4: This allows to move a large immediate value to a register. Constraint '`i`' is predefined and represents an immediate value of arbitrary size, which is limited only by the size of the register it is moved into (32 bits). Two assembler instructions are output: '**`mvkl`**' followed by a '**`mvkh`**'.

In order to enable the compiler to perform move operations not only on full-words, but also on half-words and quarter-words, '`mov`' patterns for the machine modes '`hi`' and '`qi`' must be provided. The current implementation contains such definitions, which are largely similar to the '`movsi`' pattern discussed in this section. To learn more about the '`movhi`' and '`movqi`' patterns, please refer to their definitions and the accompanying comments in 'config/c6x/c6x.md'.

3.2.3. Conditional Jumps

Conditional jumps are used in various places in the compiler. Implementing them is required for generating RTL for C statements like '`if`', '`while`', '`do {...} while`', '`for`', and '`switch`'. This section will explain the definitions in 'c6x.md' that had to be implemented in order to support statements like the above ones. First, however, it shall be explained which GCC functions are involved when a typical '`if`' statement is to be expanded to RTL code. The discussion will be rather coarse so that reading the source code in parallel to the explanations is recommended.

The first function that is invoked in the expansion of '`if`' statements is '`genrtl_if_stmt`'. Before emitting the instructions of the '`THEN_CLAUSE`' and the '`ELSE_CLAUSE`', it calls function '`expand_start_cond`' in order to emit a jump skipping the '`THEN_CLAUSE`' if the condition turns out to be false. Apart from entering a new nesting level, '`expand_start_cond`' delegates the task to function '`do_jump`', which looks at the conditional expression and decides which kind of comparison is appropriate for handling it. To this end, it contains a large '`switch`' statement with cases for the various expression types the parser may have generated. An example would be '`LT_EXPR`' (lower than), which is generated from expressions like '`a < b`'. When '`do_jump`' sees such an expression it will jump to the '`LT_EXPR`' case and call '`do_compare_and_jump`', passing it the RTL codes '`LT`' and '`LTU`', which exist for signed and unsigned 'lower than' comparisons. Function '`do_compare_and_jump`' calls '`expand_expr`' to emit instructions for computing the values of '`a`' and '`b`' and to return RTL expressions for them. The RTX representing the two results are then passed on to ‘`do_compare_rtx_and_jump`’ and from there on to ‘`emit_cmp_and_jump_insns`’, which finally passes them on to ‘`emit_cmp_and_jump_insn_1`’. The latter function is the one to actually emit RTL code. To this end, it uses the ‘optab tables’ (see Section 2.3.12) for looking up appropriate instruction codes to perform a comparison operation (‘`cmp_optab`’) followed by a conditional jump (‘`bcc_gen_fctn`’).

As can be seen from the optab patterns (‘`char *optabs[]`’ in ‘genopinit.c’), the entries of **‘`cmp_optab`’** correspond to the expander definitions for the **‘`cmpM`’** standard name, which is explained in [7] at node ‘Standard Names’. The ‘`cmpM`’ patterns accept two RTX arguments, which must represent values that shall be compared. The idea is that the RTL generated by these patterns will later produce assembler code that will set a ‘condition code register’ containing all the necessary information required for a subsequent conditional jump of arbitrary kind, which could be ‘branch if lower than’, ‘branch if greater than’, or ‘branch if equal’, for example. The bits in a condition code register typically comprise at least a zero-flag, a carry-bit, an overflow-flag, and a sign-bit. This may vary on different architectures, however.

The initialization of **‘`bcc_gen_fctn`’** for conditional jumps (standard names **‘`bCOND`’**) is performed according to another optab pattern in ‘`char *optabs[]`’, which specifies that the entries of this table should be filled with references to any defined expander definitions named ‘`blt`’ (branch if lower than), ‘`bgt`’ (branch if greater than), ‘`beq`’ (branch if equal), etc. The full list of supported conditions can be deduced from file ‘rtl.def’ by looking for RTL definitions with a ‘`'<'`’ in their last argument, which indicates the RTX class ‘comparison’. (For more information, search for ‘`'<'`’ in ‘genopinit.c’ and see the explanations at end of Section 2.3.12.)

According to the discussion of ‘`bCOND`’ in [7] at node ‘Standard Names’, any use of ‘`bCOND`’ is always immediately preceded by a use of ‘`cmpM`’ (or ‘`tstM`’)[8]. So there are two phases: compare (1) and conditionally branch (2). The way it is implemented in GCC, this two-phase model fits well with CPU architectures having a multiple-bit condition code register.

The problem with ‘C6x’, however, is that it does not know such registers. Instead, it uses condition registers that can only hold a simple truth value—using non-zero values for ‘true’ and zero for ‘false’. Further, ‘C6x’ does not support a general-purpose ‘`CMP`’ instruction, but offers special-purpose instructions like ‘`CMPEQ`’, ‘`CMPGT`’, ‘`CMPLT`’, etc., usable for initializing a condition register. If GCC were to support this scheme, there would have to be a way of telling the ‘`cmpM`’ pattern the type of condition it should use. However, there is no such way, since the only arguments accepted by the ‘`cmpM`’ pattern are the two operands to be compared.

The solution to this problem, which is suggested in [7] for ‘`bCOND`’, is to not let the ‘`cmpM`’ pattern generate any RTL instructions, but to only let it save the operands into global variables for later use by ‘`bCOND`’. As explained earlier, ‘`bCOND`’ is always preceded by ‘`cmpM`’ and will therefore always find the operands in these variables up to date. Hence, ‘`bCOND`’ has access to any information that is required for generating a special-purpose compare instruction followed by a conditionalized branch. It should therefore be the right place to emit these instructions.

Note that in order for this to work, GCC must not rely on ‘`cmpM`’ alone to have any effects on code generation. Instead, it can only expect any code to be emitted, if the use of ‘`cmpM`’ is

[8] As can be seen from the GCC source, the ‘`bcc_gen_fctn`’ table is only used in functions ‘`emit_cmp_and_jump_insn_1`’ and ‘`do_store_flag`’. For both of them this statement is true.

followed by some other pattern like ‘`bCOND`’, which uses its results. Since [7] states that ‘`cmpM`’ can be implemented as a dummy pattern, this seems to be the case. The GCC source is rather complex at this point, but seems to support this assumption.

The implementation of the different ‘`bCOND`’ patterns in ‘c6x.md’ is currently almost identical. The only difference between their definitions is the name and the corresponding RTX code, which is passed to function ‘`c6x_expand_compare`’. We shall therefore discuss only a single representative of the ‘`bCOND`’ family, which will be ‘branch if **l**ower or **e**qual’:

```
(define_expand "ble"
  [(parallel [(set (pc)
               (if_then_else (match_dup 1)
                             (label_ref (match_operand 0 "" ""))
                             (pc)))
               (clobber (match_scratch:SI 2 "" ))])]
  ""
  "operands[1] = c6x_expand_compare (LE, VOIDmode);")
```

The scenario for expanding ‘`ble`’ is as follows: First, the preparation statement is executed, which calls ‘`c6x_expand_compare`’. This function will emit the following RTL instruction:

`(SET (REG:BI` ⟨*condition reg num*⟩`) (`**`GT`**`:BI` ⟨*op0*⟩ ⟨*op1*⟩`))`

This instruction represents an assignment to a condition register (marked by machine mode ‘`bi`’). The source operand has code ‘`GT`’, which is of class ‘`<`’ (i.e. comparison). In addition, it has itself two operands representing the values to be compared. The instruction contains all the information needed by insn pattern ‘`*cmp_gt`’ for composing a complete ‘`CMPGT`’ assembler instruction. We shall later come back to ‘`*cmp_gt`’. The reason the original ‘**`LE`**’ was changed to ‘**`GT`**’ is that ‘`c6x_expand_compare`’ had to reverse the condition, because of missing support for a ‘`CMP`**`LE`**’ mnemonic on ‘C6x’. Now we shall look at the return value of function ‘`c6x_expand_compare`’, which is:

`(`**`EQ`**`:BI` ⟨*condition reg*⟩ `(CONST_INT 0))`

This expression represents a test of whether ⟨*condition reg*⟩ is equal (‘`EQ`’) to zero. The form of the test has been chosen so that it can be translated to a conditional execution construct, which is supported on ‘C6x’ by writing ‘**`[!`**⟨*condition reg*⟩**`]`** ⟨*instruction*⟩’. The return value is assigned to ‘`operands[1]`’, which will be substituted into the RTL template of the ‘`ble`’ pattern at the position of ‘`(match_dup 1)`’. Operand 0, which is passed to the ‘`ble`’ pattern as ⟨*target*⟩ of the conditional jump, will be substituted into the template at ‘`(match_operand 0 "" "")`’. The ⟨*target*⟩ is typically a ‘`code_label`’ generated by ‘`gen_label_rtx`’. The resulting RTX will look as follows:

```
(parallel [
  (set (pc)
       (if_then_else (EQ:BI ⟨condition reg⟩ (const_int 0))
                     (label_ref ⟨target⟩)
                     (pc)))
  (clobber (scratch:SI))
])
```

Since the test is ‘`EQ`’ here, insn pattern ‘`*b_false`’ will match. (An ‘`NE`’ test for a non-reversed condition would have been matched by pattern ‘`*b_true`’.) The current implementation of ‘`*b_false`’ is the following:

```
(define_insn "*b_false"
  [(set (pc)
    (if_then_else (EQ:BI (match_operand:BI 0 "general_operand" "B,A")
                         (const_int 0))
              (label_ref (match_operand 1 "" ""))
              (pc)))
    (clobber (match_scratch:SI 2 "=&b,a"))]
  ""
  "@
   [!%0] mvkl .S2 %1, %2 \;[!%0] mvkh .S2 %1, %2 \;[!%0] b .S2 %2 \;nop 5
   [!%0] mvkl .S1 %1, %2 \;[!%0] mvkh .S1 %1, %2 \;[!%0] b .S1 %2 \;nop 5"
  [])
```

Note that the constraints force operand 0 into a condition register, so that no invalid assembler code can be generated. Operand 1 will appear as a label reference (e.g. ‘`L5`’) in the assembler output.

The ‘`(clobber (scratch:SI))`’ represents the fact that this instruction will require a temporary register, which may potentially be changed (= “clobbered”) by any of the generated assembler instructions. The ‘`match_scratch`’ in ‘`*b_false`’ therefore has a ‘`&`’ constraint, which tells ‘reload’ that it is dealing with a so-called ‘earlyclobber’ operand. This indicates that no assumptions must be made as to when the operand is first modified in the sequence of assembler instructions and that it must therefore be held in a separate register[9]. The ‘`=`’ has its normal meaning, indicating to reload that the operand is of type ‘output’ (even though its value is never used after the instruction) and may therefore be modified by the instruction. The register that ‘reload’ allocates for the ‘`match_scratch`’ will be available from the first to the last assembler instruction and can be referred to by writing ‘`%2`’ in the output template.

Assuming that the compiler chooses the first constraint alternative, the sequence of assembler instructions resulting from this pattern will have the following form:

```
[!⟨cond reg⟩] mvkl .S2 ⟨target⟩, ⟨temp reg⟩
[!⟨cond reg⟩] mvkh .S2 ⟨target⟩, ⟨temp reg⟩
[!⟨cond reg⟩] b    .S2 ⟨temp reg⟩
nop 5
```

This sequence conditionally jumps to ⟨*target*⟩ if ⟨*cond reg*⟩ is zero and therefore does exactly what is expected of the ‘`*b_false`’ pattern. The second alternative would have done the same, but using ‘`A`’ registers and executing on unit ‘`.S1`’ rather than using ‘`A`’ registers and executing on unit ‘`.S2`’.

Finally, we shall revisit the RTL instruction for the comparison emitted earlier by function ‘`c6x_expand_compare`’:

[9] By default, ‘reload’ assumes that no output operand will be clobbered before all of the input operands have been used and are no longer needed. This is important for certain aspects of how ‘reload’ will combine reload registers for input and output.

`(SET (REG:BI` ⟨*condition reg num*⟩`) (`**`GT`**`:BI` ⟨*op0*⟩ ⟨*op1*⟩`))`

Its purpose was to initialize ⟨*cond reg*⟩ for the conditional branch. As already briefly mentioned, the instruction pattern to match this RTX is '∗`cmp_gt`':

```
(define_insn "*cmp_gt"
  [(set (match_operand:BI 0 "general_operand" "=b,=a")
        (gt:BI (match_operand:SI 1 "general_operand" "b,a")
               (match_operand:SI 2 "general_operand" "b,a")))
  ]
  ""
  "@
   cmpgt .L2 %1, %2, %0
   cmpgt .L1 %1, %2, %0"
  [])
```

Here again, there is a separate 'alternative' for each data path. The constraints require that either all operands are in '`b`' registers or all operands are in '`a`' registers. The alternative for '`b`' registers will generate a compare instruction to be executed on '`.L2`', whereas the '`a`' register version will be executed on '`.L1`'. As usual, the target operand is marked by a '`=`', which is important for the reload pass. The '`gt`' in the pattern helps to distinguish it from its siblings, which have been named '`*cmp_eq`', '`*cmp_lt`', '`*cmp_gtu`', and '`*cmp_ltu`', corresponding to the mnemonic they implement.

What is interesting to note is that the constraints of the target operand can be chosen as '`b`' and '`a`' rather than '`B`' and '`A`', which may be unexpected, because this operand will be used again by '∗`b_false`', whose constraints will require it to be a *condition register*. So it would have been logical to make sure it gets allocated to the right register class right from the start. It turns out, however, that it is sufficient to use '`B`' and '`A`' only in '∗`b_false`'. The reason is that when the same pseudo register is used for different operands of which one has a '`B`' and the other has a '`b`' constraint, the register class preferencing pass (function '`regclass`') will find the smaller one of both classes to be better for this pseudo register, and therefore put the pseudo into '`B`', if possible[10]. This way, the same register can be used as target operand of the '`cmpgt`' and as condition register in '`[!`⟨*cond reg*⟩`] b`'. Had the register been put in a '`b`' (but non-'`B`') register, an additional reload would have had to be output prior to the branch instruction.

Note that the currently implemented handling of conditional jumps is not the only one supported by GCC. Several other ways are offered. For example, on machines with a condition code register, '`(cc0)`' could be used for representing the results of compare instructions. However, this shall not be discussed here. For more information on this matter, please refer to [7] at node 'Jump Patterns'.

[10]Actually, '`regclass`' just computes costs (array '`costs`') and preferences (array '`reg_pref`'). The actual allocation of pseudos to hard registers is left to the register allocation pass.

3.2.4. Jumps

The pattern for unconditional jumps to a given label is called '`jump`'. Due to problems with allocating a scratch register (using '`match_scratch`'), its implementation is currently incomplete:

```
(define_insn "jump"
  [(set (pc) (label_ref (match_operand 0 "" "")))]
  ""
  "b %l0 \;nop 5"
  [])
```

The pattern generates valid assembler code most of the time, but fails when the jump ranges become too large. This is because the size of the immediate operand, which the '`b`' instruction on 'C6x' accepts as range of the jump, is limited to 21 bits width (gets left-shifted by two bits). Therefore, jumps that have ranges greater than describable in 23 bits will be output, but will result in invalid input to the assembler.

GCC knows another type of unconditional jump, which is used when the target address is given as absolute value. This pattern is called '`indirect_jump`', presumably because it has its origins in jumping to an address determined by a word in memory. However, it can be used with the target address given in a register as well. The current implementation is the following:

```
(define_insn "indirect_jump"
  [(set (pc) (match_operand:SI 0 "general_operand" "b,?a"))]
  ""
  "@
   b .S2 %0 \;nop 5
   b .S2 %0 \;nop 5"
  [])
```

Note that both alternatives must use '`.S2`', because this is the only unit to accept a jump with a *register* as target. The '`?`' in the constraints slightly discourages 'reload' from choosing the second alternative. If both alternatives would otherwise be equally costly in terms of the number of reload instructions to emit, the first one will be chosen. Currently, the '`?`' is not essential, because the branch instruction, even though being executed on the 'B' side (on '`.S2`'), is able to use the cross path for obtaining register values from the 'A' file as well. However, it might become important when support for the parallel execution features of 'C6x' is to be implemented at some time in the future.

Another type of unconditional jump can occur in '`switch`' statements. The corresponding pattern is called '`tablejump`'. As stated in [7] at node 'Standard Names', the pattern just needs to implement an absolute jump to its operand 0, if macro '`CASE_VECTOR_PC_RELATIVE`' evaluates to zero. Operand 1, which is a label immediately preceding the jump table, has been incorporated in the RTL in order to prevent the table from being removed as unreachable code. The pattern is currently defined as follows:

```
(define_insn "tablejump"
  [(set (pc) (match_operand:SI 0 "general_operand" "b,?a"))
   (use (label_ref (match_operand 1 "" "")))]
  ""
  "@
   b .S2 %0 \;nop 5
   b .S2 %0 \;nop 5"
  [])
```

3.2.5. Function Calls

Function calls require expander definitions for the '`call`' and '`call_value`' patterns to be provided. The '`call_value`' pattern is used for returning values that fit in a register, whereas the '`call`' pattern is used when either no return value is needed, as is the case with C functions returning '`void`', or when '`struct`' values larger than a register are to be returned.

The '`call`' pattern is expanded by macro '`GEN_CALL`' in function '`emit_call_1`'. The reason for not calling '`gen_call`' directly but using a macro is to be able to implement the '`call`' pattern with a varying number of arguments in different machine descriptions. The definition of '`GEN_CALL`' in 'insn-flags.h' is generated by function '`gen_macro`' in 'genflags.c', which respects the number of operands used by the '`call`' pattern in the '.md' file and generates the definition accordingly. Currently, '`call`' accepts only one argument, which is the function to call. The corresponding RTX has the following form:

```
(mem:QI (symbol_ref:SI ("⟨function name⟩")))
```

The task of expanding the call is currently completely delegated to '`c6x_expand_call`' in the preparation statements of the '`call`' pattern:

```
c6x_expand_call (NULL_RTX, operands[0]);
DONE;
```

Function '`c6x_expand_call`' works in several steps: First, it generates RTX for a new code label (return label) and a new pseudo register. Next, it emits an instruction for moving the address of the label to the pseudo. Doing so before the label has actually been emitted does not pose a problem, because only the name of the label will be output, which can later be resolved by the assembler. The next instruction that is emitted pushes the value of the pseudo onto the stack, so that the callee can use it as return address. After this, the actual call instruction is emitted, followed by the return label, which must come *after* the call. The RTX of a call instruction currently has the following form on 'C6x':

```
(call (mem:QI (symbol_ref:SI ("⟨function name⟩"))) (const_int 0))
```

This is matched by pattern '`*call_internal`', whose constraints cause the address of the function to be reloaded to a register, which is then used for generating the following two assembler instructions, which implement a jump to the callee:

```
b ⟨register⟩
nop 5
```

For calls returning a value, the compiler will use the '`call_value`' pattern (see at '`GEN_CALL_VALUE`'). The function to call will be passed in operand 1. Operand 0 will contain the hard register to return the value in. The implementation of such calls is very similar to that of calls without return value. In fact, the expansion of '`call_value`' is implemented by '`c6x_expand_call`', the same function as was used for '`call`', above. If a return value is required, however, the generated RTX for the call is slightly modified. It will be of the following form:

$$\texttt{(set } \langle \textit{return value} \rangle \texttt{ (call ...))}$$

3.2.6. Arithmetic Operations

The obvious use of patterns for arithmetic operations lies in the generation of code for the evaluation of expressions involving operators such as '+', '∗', '&', '&&', and '>>'. A not so obvious but yet important use of these patterns is that for internal purposes by the compiler. For example, the 'add' operation is used internally whenever code for adding stack frame offsets to some base register must be generated, which is quite common. This operation is therefore vital to a properly working compiler. Other patterns may not be as important. For example, experiments have shown that the pattern for 'xor' does not seem to be necessary unless the 'ˆ' operator is used somewhere in the C input file. The current implementation contains only essential arithmetic pattern definitions. More patterns would be required for being able to translate C programs containing arbitrary operators. However, since arithmetic definitions are very similar to each other, more patterns should be relatively easy to implement, based on those already available:

Pattern	**C-Operator**	**Internal Purpose**
`addsi3`	'+'	This is used for generating RTL code for stack frame offset computations and adjustments of the stack pointer.
`ashlsi3`	'<<'	This is used in the first step of emulating the extension of integer values from machine modes smaller than a word to a full word. Normally, the '`extendMN2`' pattern can be defined to support extension, but when it is not, the compiler knows[11] how to use shift patterns to achieve the same effect. See below for an explanation.
`ashrsi3`	'>>'	This is used in the second step of emulating '`extendMN2`', if the value to be extended shall be regarded as *signed*.
`lshrsi3`	'>>'	This is used in the second step of emulating '`extendMN2`', if the value to be extended shall be regarded as *unsigned*.

Currently, only patterns for the '`si`' mode (single integer) have been implemented. Patterns for smaller modes should only be defined if the target architecture has corresponding machine instructions to implement them. On RISC machines, only '`si`' mode patterns are commonly defined. The smaller and wider modes typically have to be dealt with by means of emulation. For

[11] See at '`LSHIFT_EXPR`' and '`RSHIFT_EXPR`' in function '`convert_move`'. In order to extend a byte value to a word (4 byte), it is shifted left by 24 bits and then right by 24 bits.

example, GCC has mechanisms for widening operands with a small machine mode, if the optab tables do not support the requested operation in this mode. It is therefore possible to expand an ‘add’ operation for ‘`hi`’ mode (half-integer) operands, even when no ‘`addhi3`’ pattern is defined. The operands will be widened to ‘`si`’ mode and the ‘`addsi3`’ pattern will be used instead. Please refer to function ‘`expand_binop`’ at the last ‘`for`’ loop in order to learn more on how this is realized in GCC.

As an example of how to implement arithmetic patterns, the definition of ‘`lshrsi3`’ shall be quoted here from ‘`c6x.md`’:

```
(define_insn "lshrsi3"
  [(set (match_operand:SI 0 "general_operand" "=r,r")
        (lshiftrt:SI (match_operand:SI 1 "general_operand" "r,r")
                     (match_operand:SI 2 "general_operand" "r,J")))]
  ""
  "@
   shru %1, %2, %0
   shru %1, %2, %0"
  [])
```

This is a combination of expander definition and instruction pattern. Regarding its properties as an instruction pattern, there are two template variables using ‘`general_operand`’ as predicate function. This allows almost any kind of operand to match at this position. The ‘`shru`’ instruction requires its operands[12] to be either

- ‘⟨*reg*⟩, ⟨*reg*⟩, ⟨*reg*⟩’ or
- ‘⟨*reg*⟩, ⟨*5-bit unsigned constant*⟩, ⟨*reg*⟩’

where the first is the source operand, the second is the amount to shift it by, and the third is the target operand the result is written to. To implement support for both options, two ‘alternatives’ were introduced.

	Number of Alternative	
	0	**1**
Operand 0	r	r
Operand 1	r	r
Operand 2	r	J

As mentioned before, each ‘`r`’ constraint forces its operand to a register. The ‘`J`’ constraint is defined by macro ‘`CONST_OK_FOR_LETTER_P`’ in ‘`c6x.h`’. Because of this constraint, the second alternative will only be chosen if operand 2 has RTX code ‘`CONST_INT`’ and has a value in the range from 0 to 31. Since the assembler output strings of both alternatives do not differ, the pattern might also have been implemented using just *one* alternative, as follows:

```
(define_insn "lshrsi3"
  [(set (match_operand:SI 0 "general_operand" "=r")
        (lshiftrt:SI (match_operand:SI 1 "general_operand" "r ")
                     (match_operand:SI 2 "general_operand" "rJ")))]
  ""
  "shru %1, %2, %0"
  [])
```

[12] Capabilities relating to 40-bit operands have not been considered in this work.

Even though slight semantic differences cannot be completely excluded, this implementation should be largely equivalent to the one above. Writing several constraint letters in the same alternative tells GCC that an operand matched by either of them would directly be acceptable for this alternative.

3.3. Adding ‘C6x’ to the GCC Make Files

Before any ‘`cc1`’ (C compiler) binary can be built from the GCC sources, the ‘configure’ shell script included in the source tree has to be run. This will generate the makefiles needed to translate the GCC sources so that support for a specific target will be enabled. The target can be specified to ‘configure’ with option ‘`--target=`⟨*target*⟩’.

Before GCC can be built for the ‘C6x’ architecture, its build environment should be modified so that ‘`--target=c6x`’ will become a valid command line option. This can be achieved by making the following modifications[13]:

- ‘config.sub’:
  - Add ‘`| c6x-*`’ in the line containing ‘`| c4x-* | c54x-*`’.
  - Add ‘`| tic6x-*`’ in the line containing ‘`| tic4x-* | tic54x-*`’.
  - Add the following block after the ‘`tic54x | c54x*)`’ block:
```
tic6x | c6x*)
    basic_machine=tic6x-unknown
    os=-coff
    ;;
```

- ‘configure.in’:
  - Add the following block after the ‘`c54x*-*-* | tic54x-*-*)`’ block:
```
c6x-*-* | tic6x-*-*)
    noconfigdirs="$noconfigdirs ${libstdcxx_version} \
                  target-libgloss ${libgcj}"
    ;;
```

- ‘gcc/config.gcc’:
  - Add the following block after the ‘`tic4x-*-*)`’ block:
```
tic6x-*-*)
    cpu_type=c6x
    ;;
```
  - Add the following block after the ‘`c4x-* | tic4x-*)`’ block:
```
c6x-* | tic6x-*)
    ;;
```

[13]File names are meant relative to the GCC source tree, not relative to ‘gcc/’, here.

- ‘gcc/Makefile.in’:
  - Replace ‘`LIBGCC = libgcc.a`’ with ‘`LIBGCC =`’ in order to prevent GCC from trying to build ‘libgcc’ (target support routines) for ‘C6x’, which would require a fully working cross-compiler.

After that, all files of the machine description (‘c6x.h’, ‘c6x.md’, ‘c6x.c’, and ‘c6x-protos.h’) should be moved to directory ‘gcc/config/c6x/’. Finally, GCC can be configured, typing the shell command ‘`sh> ./configure --enable-languages="c" --target=c6x`’, and can be built by typing ‘`sh> make`’. If the build succeeds, the C compiler will be available as ‘gcc/cc1’.

4. Conclusion and directions for further work

This work has discussed some fundamental concepts of the GCC internals and how this knowledge can be used to implement support for a new target architecture. Based on this, a machine description was implemented, providing basic support for ‘C6x’ as a GCC target. During the different stages of the implementation, several cross-compiler versions were built and used for performing compilation tests on C sample programs in order to ensure the proper function of the compiler for a reasonable subset of the C language.

Building on what has been achieved, further work might aim at improving and completing the current implementation. A number of suggestions shall be made here in this respect:

- Support for the complete set of C **operators** should be provided. This would require to implement a number of arithmetic expander definitions and instruction patterns such as for the standard names ‘`andM3`’, ‘`xorM3`’, ‘`mulM3`’, etc.
- **Floating point** types and operations should be supported on the ‘C6x’ target. The following GCC source files might be relevant for the emulation of floating point functionality: ‘real.c’, ‘floatlib.c’, and ‘config/fp-bit.c’.
- A ‘C6x’ **assembler** and any other required parts of the toolchain would have to be implemented in order to have a means for generating binary code.
- The conventions for **function calls** (which arguments to pass in registers etc.) and the layout of types in memory should be made compliant with those described in Chapter 8 of [15]. This might open the way for linking GCC-compiled object code against object code compiled by ‘Texas Instruments TMS320C6000 Code Composer Studio’.
- The implementation of the **‘`jump`’ pattern** (Section 3.2.4) should be fixed so that correct code for large distance jumps can be generated. Function ‘`get_attr_length`’ in connection with ‘`shorten_branches`’ might provide the basis for this. Different output templates might be implemented for providing a short and long form of branch instruction, which could be selected by the value of the ‘`length`’ attribute.
- Support for conditional execution could be added using the mechanisms described in [7] at node ‘Conditional Execution’. This would allow GCC to conditionalize any kind of instruction and to dispense with some of the conditional jumps currently emitted for implementing ‘`if`’ statements.
- In load and store operations, the ‘C6x’ architecture supports an addressing mode where the combination of two registers forms the address of a memory location to access. The first one is the base register, the second one is the index register to be scaled by the number of bytes accessed by the current instruction, and to be added to the first one. For example, ‘`ldh *A10[A2], A1`’ would load a **2**-byte value (half-word) from the memory location pointed to by ‘`A10`’ $+ 2 *$ ‘`A2`’ into ‘`A1`’. It might be desirable to be able to generate this kind of address in order to save some of the instructions currently needed for performing address computations.

- 'Delay slot scheduling'. On 'C6x', there are five delay slots until the code at the target of a jump is actually executed. Currently, these slots are filled with '`nop`'. Delay slot scheduling would allow GCC to fill them with useful instructions.

- Implementing 'VLIW instruction packing' might yield another performance win. This is a technique in which several instructions are scheduled for execution in the same processor cycle. File 'genautomata.c' contains a comment suggesting that some of the required support mechanisms are in principle already available in GCC.

- Switching between little- and big-endianness could be implemented.

- The testcases beneath the 'testsuite/' directory might be used to get hints as to which functionality still has to be implemented in order to get a fully functional compiler.

- Other interesting issues might be: generation of debugging output, support for code profiling, generation of 'position independent code' (PIC), support for auto-increment and auto-decrement addressing, and support for 'sibling calls'.

- In order to support C++ as an input language, macros for 'exception handling' and 'multiple inheritance' would have to be implemented.

A. Terms and Abbreviations

ABI GCC → 'Application Binary Interface'

Absolute Declarator GCC
: A type specification as you would write it in a '`typeof(⟨...⟩)`' or a typecast expression. For example: '`typeof(int (*)(int a)) fp1, fp2;`'. References: non-terminal symbol '`absdcl`' in 'c-parse.y'.

Access Link GCC
: Used in [1] as a synonym for → 'static chain pointer'.

Activation Record GCC
: A data structure created on the stack when a C function is being called. Among other things it contains slots for local variables, temporaries, the actual function arguments, and the → 'control link' (also known as → 'dynamic link') to the activation record of the calling function. References: Section 3.1.3 and [1] on page 398.

Aggregate Type GCC
: A C type, which is too large to be held solely in registers. When passing aggregate types as function parameters or as return values, GCC actually passes pointers to them, rather than the values themselves. Typical examples of 'aggregate types' are structs with a → 'machine mode' larger than '`DImode`'. References: function '`aggregate_value_p`'.

Alignment Chain GCC
: The alignment chain of an instruction is used during 'branch shortening' to determine alignment requirements. For further explanation, please refer to the comment at the definition of '`struct label_alignment`'. References: function '`compute_alignments`'.

Alternative GCC
: An 'insn pattern' can support several alternatives, which means that the → 'constraints' in its '`match_operand`' → 'template variables' are comma separated lists, in which each element specifies the operand's constraint for one alternative. This is used, for example, in the pattern for move instructions: Moves from memory (constraint '`m`') to a register (constraint '`r`') should generate a load instruction, whereas moves from a register to memory should generate a store. This difference can be described using two alternatives with the appropriate constraints.

AMR TI Addressing mode register

AP GCC → 'Argument pointer'

Application Binary Interface GCC
: Defines the conventions for calling high-level language functions from within Assembler modules. This is important when interfacing with system libraries or when connecting programs written in different languages. Normally, there is only one ABI for an operating system running on a certain machine architecture. An ABI typically includes specifications for the following aspects:
    - memory layout of variables of certain C types, e.g. the bit width of an '`int`', a '`long int`', etc.

- padding in structs (for better alignment)
- parameter passing on the stack (in which order, how to align, which parameters to pass in registers)
- returning of function values

Argument Pointer GCC
: A hard register, which on some → 'machines' is used as a pointer to incoming actual function parameters on the stack. Often, its use is deactivated by defining its register number equal to that of the → 'frame pointer' register. If it is active, however, it must either be member of '`FIXED_REGISTERS`' and have a corresponding existing CPU register for this purpose, or it must be eliminable (most often to → 'FP') my means of → 'register elimination'. References: [7], node 'Frame Registers' at '`ARG_POINTER_REGNUM`'; GCC source at '`arg_pointer_rtx`'.

Assembler Output Template MD
: → 'output template'

Automatic Variable GCC
: A variable local to a function, whose storage is automatically released on function exit.

BFD Library
: Binary file descriptor library. Mainly reads, translates, and writes object file formats and handles relocations. Used, for example, by 'gas' and 'ld'.

Binding Contour GCC
: A scope in C such as enclosed by a {. . . }-block. For example, a function body has an associated binding contour. References: '`struct binding_level`'; functions '`pushlevel`' and '`poplevel`'.

Binding Level GCC
: A synonym for → 'binding contour'.

biv GCC A basic induction variable in loop optimization. References: 'loop.c'

Block Stack GCC
: The stack of → 'binding contours'. References: comment at definition of '`struct nesting`'.

BTB TI Branch target buffer

Body, of an Insn GCC
: The main part of an RTL instruction. Describes the effects of this insn (which operator and operands to use, where to place the result). The RTL of the body can be accessed by macro '`PATTERN(`⟨*insn*⟩`)`'.

Branch Shortening GCC
: Compiler pass, which determines (mostly on RISC machines) whether the range of each jump instruction is short enough to use the shorter sequence of machine instructions for performing the jump (if such sequence exists). For example, a short form might be a single instruction allowing to specify the range as an immediate value, whereas a long form would require to load the range into a register and then use an extra insn for actually jumping. References: [7], node 'Passes' at 'Branch shortening'; function '`shorten_branches`'; function '`get_attr_length`'.

Build Machine GCC
: The 'machine' a 'gcc' binary is built on. If using a → 'cross-compiler', this will be different from the machine 'gcc' will later be run on. References: [7], node 'Configure Terms'.

Build-Time GCC
: The time when the source files of the GCC distribution are being translated to a 'gcc' binary. See also: → 'compile-time' and → 'runtime'.

Byte GCC In the context of RTL, a 'byte' is a synonym for → 'unit', which refers to an object of 'BITS_PER_UNIT' bits width.

C6x TI Texas Instruments TMS320 C6000 series of DSPs (→ 'DSP')

Call Chain GCC
: Let function 'A' call 'B', which calls 'C'. The corresponding call chain would be 'A' $\xrightarrow{\text{calls}}$ 'B' $\xrightarrow{\text{calls}}$ 'C'.

Caller Save Area GCC
: Stack slots which a caller of a function allocates in its → 'stack frame' in order to save registers that might be clobbered (→ 'clobber') by the callee. The caller can treat these registers as if they remained 'live' across the call, because it will restore them from the saved values as soon as the callee returns.

Calling Sequence GCC
: The sequence of instructions the compiler must emit in order to perform a classic function call. This typically involves pushing the function arguments and the return address onto the stack, as well as allocating a new → 'stack frame' for the called function. References: [1] at page 404–408.

Canadian Compiler GCC
: A compiler where → 'build machine', → 'host machine', and → 'target machine' are all different. References: [7], node 'Configure Terms'.

CC GCC → 'Condition code'

ce GCC → 'Conditional execution'

CF Carry flag

CFA GCC Canonical frame address

CFG GCC → 'Control flow graph'

clobber GCC
: If a register is clobbered, its value gets overwritten and is no longer available. When loading some value to a register, the compiler makes sure that the register will not be clobbered before having been used for the intended purpose. To this end, it needs precise knowledge of which RTL insns can potentially clobber a register. In 'set' insns, it is obvious that the target will be clobbered. However, there are other instances where a clobber is there, but the RTL does not indicate it. In these cases, an explicit 'clobber' expression must be used to describe the side-effect. References: [7], node 'Side Effects'.

COFF TI Common object file format

Compiler Runtime GCC

Has the same meaning as → 'compile-time'.

Compile-Time GCC
: The time when user source files are being compiled using an existing 'gcc' binary. This is sometimes also called → 'compiler runtime'. See also: → 'build-time' and → 'runtime'.

Completion Order GCC
: A synonym for post-order in traversals of binary trees.

Cond Exec GCC
: → 'Conditional execution'

Conditional Execution GCC
: Nullifying machine instructions depending on the value of a previously initialized condition register. References: [7], node 'Conditional Execution'; functions 'cond_exec_∗' and 'noce_∗' in 'gcc/ifcvt.c'.

Condition Code GCC
: A special CPU register that exists on some architectures to hold the result of 'compare' and 'test' instructions, which can be used by a following conditional branch instruction. References: [7], node 'Jump Patterns'.

Constant Pool GCC
: Some CPU instructions on some architectures do not allow immediate values as arguments, but only memory references. As a consequence, in situations where a function needs to pass an immediate to such an instruction, the immediate must be available as an addressable copy in memory. To this end, GCC outputs Assembler directives that define and initialize a memory area, called 'constant pool', which is located directly before (or after, depending on macro 'CONSTANT_POOL_BEFORE_FUNCTION') the Assembler instruction block of the function and which will hold the corresponding immediates. References: implementation of constant pools in 'varasm.c', in particular function 'force_const_mem' and 'output_constant_pool'.

Constraints MD
: Can be used in a 'match_operand' template for telling the reload pass which kind of operand, e.g.: memory, register, or immediate, is expected by the → 'assembler output template' for this operand. If the concrete RTL instruction has an operand of a type that is different from what the constraints require, → 'reload' will prepend the RTL instruction by appropriate instructions, which, for example, move a memory operand to a register in order to satisfy a constraint requiring a register. Note that, as opposed to → 'predicates', constraints have no influence on whether an → 'instruction pattern' matches or not. The constraint does not play a role until after the predicate has matched. References: [7] at node 'Constraints', Section 2.3.10 and → 'alternative'.

Control Flow Graph GCC
: A directed graph with the 'basic blocks' of the function as nodes. The edges represent transfer of control from one block to another by means of jumps or fall-through conditions. References: [1], p. 532; GCC source files 'gcc/cfg∗.c'; function 'find_basic_blocks' as entry point.

Control Link GCC
: Used in [1] as a synonym for → 'dynamic chain pointer'.

Cross-built Native Compiler GCC
: A compiler where → 'host machine' and → 'target machine' are the same, but the → 'build machine' is different. References: [7], node 'Configure Terms'.

Cross-Compiler GCC
: A compiler where → 'build machine' and → 'host machine' are the same but the → 'target machine' is different. References: [7], node 'Configure Terms'.

CSE	GCC	Common subexpression elimination.
CSR	TI	Control status register
Ctor	GCC	Constructor

Data Flow Analysis GCC
: A compiler pass performing analysis of RTL code. Divides the program into basic blocks and finds out where variables in the user program are assigned to and where they are used. This pass also performs dead code elimination. References: [7], node 'Passes' and comments in 'flow.c' and 'df.c'.

DBR GCC → 'Delayed branch reorganization'

Defi nition of a Variable (data fbw analysis) GCC
: In [1], on page 610, the following definition is given: "A definition of a variable '`x`' is a statement that assigns, or may assign, a value to '`x`'." Definitions are either ambiguous or unambiguous:

- ambiguous definition of var '`x`':
  1. a call of a procedure with '`x`' as call-by-reference parameter or with '`x`' aliased with some entity defined at some point in the function
  2. an assignment to a pointer that might refer to '`x`'
- *un*ambiguous definition:
  1. unconditional assignment to '`x`'
  2. unconditional in/out operation storing a value into '`x`'

Delay Slots GCC
: If an instruction has n delay slots, the result it computes cannot be used by the next instruction in the next CPU cycle but only after waiting for n extra CPU cycles. For branches, n delay slots mean that n one-cycle instructions following the branch instruction will still be executed before the branch comes into effect. References: [7], node 'Delay Slots'.

Delay Slot Scheduling GCC
: Fills the → 'delay slots' of a branch instruction with instructions that would normally have to be written before the branch, so that they physically go after but still are executed before the branch. Also called → 'delayed branch reorganization' within GCC. Implemented by function '`dbr_schedule`' in 'reorg.c'. References: [7], node 'Delay Slots'; comment at the top of 'reorg.c'.

Delayed Branch Reorganization GCC
A synonym for → 'delay slot scheduling'.

Depth First Order GCC
A synonym for pre-order in traversals of binary trees.

DF	GCC	→ 'Data flow analysis'
DFS	GCC	Depth first search
DMA	TI	Direct memory access

Dominator GCC
A → 'control flow graph' node d is a 'dominator' of node n if every path from the initial node of the graph to n goes through d. This can be used in automatic loop detection. References: [1], p. 602 and '`calculate_dominance_info`' in 'dominance.c'.

DSK	TI	DSP starter kit
DSP		Digital Signal Processor
Dtor	GCC	Destructor

Dynamic Chain Pointer GCC
Pointer located in the → 'stack frame', which points to the immediate caller's stack frame, typically to the caller's 'dynamic chain pointer'. Dereferencing these pointers allows to follow the → 'call chain' from the innermost function to function '`main`'.

Dynamic Link GCC
A synonym for → 'dynamic chain pointer'.

Dynamic Linker
'ld.so' loads the shared libraries needed by a program, prepares the program to run, and then runs it. Should not be confused with → 'link editor' ('linker' for short), which works at → 'compile-time' rather than program load time. References: manual page of 'ld.so'.

Dynamic Loader
A synonym for → 'dynamic linker'.

ea GCC List **e**nds with an **a**ttribute. Used in 'c-parse.y'.

earlyclobber GCC
An earlyclobber operand is an operand of an RTL instruction, which may be overwritten before the instruction is finished using its input operands. References: [7] at node 'Modifiers' and → 'clobber'.

EH	GCC	→ 'Exception handling'
EMIF	TI	External memory interface
EP	TI	Execute packet

Epilogue of a Function GCC
Automatically generated statements at the bottom of each function, which are responsible for reinstating the → 'stack frame' that was active before this function was called. This includes restoring registers, which were saved on the stack in the prologue, restoring → 'SP' from → 'FP', restoring the old FP from the stack, and finally returning control to the caller.

Exception Handling GCC

Mechanisms for handling exceptions which some languages allow to be thrown on encountering an error condition. In C++, for example, this would be used by writing a '`try`'-'`catch`'-block. References: 'except.c'

executable GCC
: If a file is executable, it is capable of being loaded into memory and run as a program.

Expander Definition GCC
: A '`define_expand`' in the → 'machine description', which provides a definition for some 'standard name' that can be used by the → 'RTL generation' pass. References: Section 2.3.11 and [7] at node 'Expander Definitions'.

Expansion GCC
: → 'RTL Generation'

Expression Code GCC
: One of the enumerators in '`enum rtx_code`', identifying the type of an RTX. All available expression types are defined in 'rtl.def'.

extv GCC **Ext**ract bit-field **v**alue. References: [7], node 'Standard Names'.

extzv GCC **Ext**ract **z**ero-extended bit-field **v**alue. References: [7], node 'Standard Names'.

FFS GCC → 'Find first set bit'

Field GCC In an '`rtx_def`', this is one element of the '`fld[]`' array corresponding to an operand of the RTX. Such elements are of type '`rtunion`'.

Find First Set Bit GCC
: Arithmetic operation returning one plus the index of the least significant 1-bit in the argument, or returning zero if the argument is zero.

Flag GCC RTL expressions contain several flags (one-bit bit-fields) that are used in certain types of expressions for indicating special properties. For instance, if an insn is part of a function prologue, it will be generated with the 'frame related' flag being set. In the textual form this would show up as '`/f`', as in '`(reg/f:SI 7)`', for example. Insn flags can be both read and written via the macros listed in [7], node 'Flags'. References: comments in definition of '`struct rtx_def`'.

Flow GCC → 'data flow analysis'

Flow Graph GCC
: → 'control flow graph'

FP TI Fetch packet

FP GCC Frame pointer

Frame Pointer GCC
: A hard register, which is typically used as base register for access to → 'automatic variables' and → 'temporary slots' in a → 'stack frame'. Note that the → 'hard frame pointer' can be implemented as a register separate from that.

Function Inlining GCC
: In an inlined function call, the instructions of the function are expanded and inserted immediately instead of generating a classic → 'calling sequence' and emitting the function separately. The code will behave as if the function had been called even though no call/return is physically performed. References: 'integrate.c'.

GCSE GCC

		Global common subexpression elimination
GCC		GNU Compiler Collection
GGC	GCC	GCC garbage collector
GIE	TI	Global interrupt enable
giv	GCC	A general induction variable in loop optimization. References: 'loop.c'
Global Offset Table	GCC	A table of pointers to the global variables of a dynamic shared library. The GOT is part of the data segment, since relocation, which is performed at program load time by the → 'dynamic linker', requires read-write access to the entries. See also: → 'Position Independent Code'. References: http://www.iecc.com/linker/linker10.html.
GOT	GCC	→ 'global offset table'
Goto Fixup	GCC	In some cases, forward '`goto`' statements cannot be expanded before the corresponding label definition is seen by GCC. When such a case occurs, the '`goto`' is put on a fixup list and will be expanded retrospectively. Fixups are needed when the block in which the target label will be output cannot be determined yet. References: comment at definition of '`struct goto_fixup`'; function '`expand_fixup`'; function '`fixup_gotos`'.
Hard Frame Pointer	GCC	If defined in the → 'machine description', the 'frame pointer' will be eliminated in favour of this register during → 'register elimination'. References: [7], node 'Frame Registers' at '`HARD_FRAME_POINTER_REGNUM`'.
Hard Register	GCC	A register that corresponds to an existing CPU register. See also: → 'pseudo register'.
Host Machine	GCC	The → 'machine' the 'gcc' binary is compiled for and can be run on. References: [7], node 'Configure Terms'.
HT	TI	Hyperthreading
ICE	GCC	Internal compiler error
IER	TI	Interrupt enable register
IFR	TI	Interrupt flag register
ILP	TI	Instruction level parallelism
Inbound Register	GCC	Same as → 'incoming register'.
Incoming Register	GCC	A register, in which a called function will find one of its parameters. There will be no incoming registers if a machine passes function arguments only on the stack. See also: → 'outgoing register'. References: [7], node 'Register Basics' at '`INCOMING_REGNO`'.
Inlining	GCC	→ 'function inlining'
Input Reload	GCC	An extra instruction emitted by the → 'reload pass' in order to load a value from

memory to a → 'reload register' so that the actual instruction can use the value as if it had been in a register right from the start.

Insn GCC Short for 'RTL Instruction'.

Instruction Code GCC
Sequential number, assigned to each → 'instruction pattern' defined in the '.md' file of the → 'machine description'. See → 'instruction recognition' for how the instruction code of an RTL instruction is recognized by GCC. References: '`enum insn_code`' in 'insn-codes.h'.

Instruction Elision GCC
The Exclusion of certain → 'instruction patterns' when reading the '⟨*arch*⟩.md' file (→ 'machine description'). Patterns are excluded when their C expression in the condition field can be determined to be false at → 'build-time'. References: variable '`insn_conditions`', function '`maybe_eval_c_test`' in file 'gensupport.c'.

Instruction Pattern MD
A '`define_insn`' in the '.md' file of the → 'machine description'. Defines a pattern which matches on RTL instructions of a certain structure. The definition tells how to instantiate any '`match_operand`' variables in the pattern from a concrete RTL instruction and how to translate the instantiated pattern to one or more Assembler statements. References: Section 2.3.9; [7], node 'Patterns'.

Instruction Recognition GCC
Macro '`recog_memoized`' can be used to match an RTL instruction against the available patterns in the '⟨*arch*⟩.md' file (→ 'machine description') and return the → 'instruction code' of the first matching pattern. (If no pattern matches, i.e. the instruction cannot be recognized, -1 is returned.) This is called instruction recognition. The '`INSN_CODE`' field of the → 'insn' gets initialized accordingly, so that the insn does not need to be recognized again the next time the instruction code is requested.

Instruction Scheduling GCC
A compiler pass which reorders instructions within a basic block, with the aim of reducing the number of pipeline stalls caused by the use of data that has been computed shortly before, but is not immediately available. Where possible, it moves the computing and the using instructions further apart from each other. References: [7], node 'Passes' at 'Instruction scheduling'.

insv GCC **Ins**ert bit-field **v**alue. NB: the 'insv' pattern actually does not insert bits into the target operand, but replaces them. References: [7], node 'Standard Names'.

intl GCC Internationalization

IRP TI Interrupt return pointer

ISFP TI Interrupt service fetch packet

IST TI Interrupt service table

Jump Optimization GCC
A compiler pass which simplifies jumps to the following instruction, jumps across jumps, and jumps to jumps. References: 'jump.c' and [7], node 'Passes'.

Jump Threading GCC
Detects a conditional jump that branches to an identical or inverse test. Such jumps

can be “threaded” through the second conditional test. References: [7], node ‘Passes’.

LCM GCC Lazy code motion. This is a special form of → ‘GCSE’.

libgcc GCC
A library of support routines that can be linked with object files on the target. Should be compiled with a ‘gcc’ binary outputting code for the target.

linkable GCC
If a file is linkable, it can be used as input by a → ‘link editor’ such as ‘ld’.

Link Editor
Combines a number of object and archive files, relocates their data and resolves symbol references, producing an → ‘executable’. When compiling a program, the link editor is usually the last step in the → ‘toolchain’. Hence, it works at → ‘compile-time’, as opposed to the → ‘dynamic linker’, which works at program load time or sometimes at → ‘runtime’. References: manual page of ‘ld’.

Linker Commonly refers to the → ‘link editor’. In some rare cases, it may also denote the → ‘dynamic linker’, but this is rather unusual. References: http://www.iecc.com/linker/linker10.html.

loadable GCC
A loadable file can be loaded into memory as a library along with a program.

Machine GCC
A machine refers to the pair (CPU architecture, operating system). GCC distinguishes three contexts in which the term ‘machine’ can be used: the → ‘build machine’, the → ‘host machine’ and the → ‘target machine’.

Machine Description GCC
Comprises several files which must be implemented when a new target architecture is to be supported, in order to tell the compiler the set of CPU features that can be used. The files should be put beneath ‘config/⟨*arch*⟩/’ and typically include a ‘⟨*arch*⟩.md’ file in RTL syntax with definitions of supported → ‘instruction patterns’, and a ‘⟨*arch*⟩.h’ target macro file containing macro definitions that influence the internal behaviour of GCC. References: [7], node ‘Machine Desc’.

Machine Mode GCC
A machine mode describes the size of an RTL data object such as a ‘`reg`’ or a ‘`mem`’. This can be QImode for ‘byte’, DImode for ‘word’, etc. Machine modes are defined in ‘machmode.def’. References: [7], node ‘Machine Modes’.

MD GCC → ‘machine description’

MI Thunk GCC
→ ‘multiple inheritance thunk’

Multilib GCC
A mechanism ‘gcc’ provides for cases where ‘libgcc.a’ (→ ‘libgcc’) must be compiled in multiple versions. For example, this is the case when a target supports a compiler option that switches between big and little endian mode. References: [7], node ‘Target Fragment’; function ‘`print_multilib_info`’; file ‘multilib.h’.

Multiple Inheritance Thunk GCC
A small function used in the implementation of multiple inheritance in C++. It adjusts

the '`this`' pointer in virtual function calls. Also called 'thunk' or 'mi thunk'. References: functions '`use_thunk`' and '`make_thunk`' in 'cp/method.c'; [2], p. 228, Section 10.8c 'Multiple Inheritance and Virtual Functions'.

Native Compiler GCC
: A compiler where → 'build machine', → 'host machine', and → 'target machine' are all the same. References: [7], node 'Configure Terms'.

noce	GCC	**No** **c**onditional **e**xecution
noea	GCC	List does **no**t **e**nd with an **a**ttribute. Used in 'c-parse.y'.
nosa	GCC	List does **no**t **s**tart with an **a**ttribute. Used in 'c-parse.y'.
nosc	GCC	**No** **s**torage **c**lass. Used in 'c-parse.y'.
nots	GCC	**No** **t**ype **s**pecifier. Used in 'c-parse.y'.
NT	GCC	Non-terminal symbol in a grammar. With 'bison', the semantic value of an NT is computed by the semantic action of the grammar rule used for the current → 'reduction'.

Object Stack GCC
: 'Libc' data structure which is used for a sequence of data items with strongly differing sizes. All items fill a contiguous memory area. At any time, only the item on top can be grown. In return, 'obstacks' support it to be grown to an arbitrary size. References: 'obstack.h'.

Obstack GCC
: → 'object stack'

Operator Table GCC
: Refers to array '`optab`', which holds information on which → 'expander definition' to use when a certain kind of operation (arithmetic, jumps, calls, etc.) occurs in the user code. The table can be used to obtain the appropriate → 'instruction code'. References: Section 2.3.12.

Optab GCC
: → 'operator table'

Outbound Register GCC
: Same as → 'outgoing register'.

Outgoing Register GCC
: A register that the caller uses to pass values to some function. There will be no outgoing registers if a → 'machine' passes function arguments only on the stack. If the → 'target machine' has → 'register windows', the number of an 'outgoing register' and the number of the corresponding → 'incoming register' may differ. References: [7], node 'Register Basics' at '`OUTGOING_REGNO`'.

Output Reload GCC
: An extra instruction emitted after an instruction by the → 'reload pass' in order for the instruction to be able to write its results to a register (the → 'reload register') and have it automatically written to memory afterwards.

Output Template MD
: A string which specifies how to output the assembler code for an → 'instruction pattern'. Can contain variable parts marked by ''`%`*char* (*digit*)?'', most of which

stand for the operands of the instruction in Assembler syntax. References: [7], node 'Output Template'.

PCC GCC Portable C compiler. See [8], node 'Code Gen Option' at '-fpcc-struct-return'; macro '`PCC_STATIC_STRUCT_RETURN`'.

PCE1 TI Program counter of current fetch packet in pipeline phase E1

PIC GCC → 'position independent code'

PLT GCC → 'procedure linkage table'

Position Independent Code GCC

PIC code is used when precisely the same program code is to be run by several processes at different virtual addresses. The most prominent example are probably dynamic shared libraries. With such libraries, references to global variables cannot use absolute addresses, since they would be wrong in at least one of two processes running at different addresses. PIC code uses indirect addressing via the → 'GOT' to solve this problem. References: http://www.iecc.com/linker/linker10.html.

Predicate MD

→ 'predicate function'

Predicate Function MD

Part of '`match_operand`' templates, which specifies the conditions that operands must meet in order to be accepted by this template. Unlike → 'constraints', a non-matching predicate makes its → 'instruction pattern' reject the current instruction immediately. If no other instruction pattern matches, the compiler will abort with an error. References: [7], node 'RTL Template' at '`match_operand`' and Section 2.3.10.

Predication GCC

A synonym for → 'conditional execution'.

Procedure Linkage Table GCC

Similar to the → 'GOT', procedure calls to globally defined functions cannot use absolute addresses, but must use the PLT as means of indirection. The PLT supports lazy procedure linkage, where the → 'dynamic linker' looks up functions only at the first time they are actually called. References: http://www.iecc.com/linker/linker10.html and → 'position independent code'.

Prologue of a Function GCC

Automatically generated statements at the top of each function, which are responsible for setting up the → 'stack frame'. This includes saving the old value of FP, saving → 'SP' in → 'FP', and saving registers, which are to be preserved, on the stack.

Pseudo GCC

→ 'pseudo register'

Pseudo Register GCC

A register that does not actually exist as CPU register, but is used during compilation as if it were a register. Pseudo registers can be generated in arbitrary quantities. However, before Assembler code can be output, they must be allocated to → 'hard registers' by the → 'register allocator' or put into a memory slot on the stack by '`alter_reg`'. A special kind of pseudo register, which is treated differently, is the

→ 'virtual register'.

qty GCC → 'quantity number'

Qual GCC See → 'type qualifier'.

Qualifi ed Type GCC
A type with one or more → 'type qualifiers'.

Quantity Number GCC
Used for representing → 'tying' in local register allocation. Tied registers get the same quantity number and are thus treated as a single register by the → 'register allocator'. References: comment at top of file 'local-alloc.c'.

Queued Expression GCC
RTL expressions of → 'expression code' '`QUEUED`' are used only during the → 'RTL Generation' pass. They are created by function '`enqueue_insn`', which is called when a post-increment or post-decrement operation is such as '`a[i--]`' is to be expanded. References: function '`protect_from_queue`'; access macros '`QUEUED_*`' such as '`QUEUED_VAR`'.

RA GCC → 'register allocator'

RC GCC Reverse completion order (→ 'Completion Order')

recognize an instruction GCC
→ 'instruction recognition'

Reduction GCC
Application of a rule in a grammar file in order to reduce a sequence of tokens and non-terminal symbols to another non-terminal. The other action that is typical for bottom-up parsers like 'bison' is called → 'Shifting'. References: [10] at node 'Algorithm'.

Register Allocator GCC
Compiler pass which allocates hard registers for → 'pseudo registers' to live in. Depending on the flag '`flag_new_regalloc`', either the classic '`local_alloc`' and '`global_alloc`', or the new graph coloring register allocator '`reg_alloc`' is used.

Register Instantiation GCC
The Conversion of RTL → 'virtual register' references to → 'hard register' references. References: function '`instantiate_virtual_regs`'.

Register Elimination GCC
Replaces any eliminable RTL register references (such as to → 'FP') with a '`plus`' of another register reference (such as → 'SP') and an offset. The offset must be determinable by the compiler after code generation (influences number of slots for temporaries (→ 'temporary slot') in the → 'stack frame') and register allocation (influences number of registers in the → 'register save area') are finished. Elimination is part of the → 'reload pass'. References: [7] at node 'Elimination' and in the GCC source the call chain '`reload`' $\xrightarrow{\text{calls}}$ '`reload_as_needed`' $\xrightarrow{\text{calls}}$ '`eliminate_regs_in_insn`', as well as function '`elimination_effects`', and function '`set_initial_elim_offsets`'.

Register Save Area GCC

A part of the → 'stack frame' which is used for saving registers in the → 'prologue of a function' and restoring them in the → 'epilogue of a function'. This allows for the caller to safely assume that these registers keep their values across function calls. Please do not confuse this mechanism with caller saves (→ 'caller save area'), which are done by the caller rather than by the callee. References: function '`c6x_expand_epilogue`' in 'config/c6x/c6x.h'.

Register Scan GCC

A compiler pass which is run before → 'CSE' and Loop Optimization and finds the first and last use of each → 'pseudo register'. References: comment before function '`reg_scan`' in 'regclass.c'

Register Spilling GCC

A → 'pseudo register' can be spilled from a hard register, depriving the pseudo of its home in the hard register, because the latter is needed elsewhere. References: comment at top of 'reload1.c'; functions '`spill_hard_regs`' and '`finish_spills`'.

Register Transfer Language GCC

The intermediate language used by GCC for representing user programs. As opposed to the → 'tree representation', RTL is more like a sequence of instructions. Unlike assembler code, it looks very similar on all → 'machines', so that the same compiler passes can be used for all target architectures and all input languages. Please refer to Section 2.3.2 on page 13.

Register Windows GCC

With register windows, the register in which a function sees an argument is not necessarily the same as the one in which the caller passed the argument. References: macros '`INCOMING_REGNO`', '`OUTGOING_REGNO`', and '`LOCAL_REGNO`'

Reload GCC

→ 'reload pass'

Reload Pass GCC

Mainly performs the following tasks:

- satisfy unmet → 'constraints' by generating → 'RTL' instructions for copying operands to other kinds of storage, e.g. from memory to a register
- renumber → 'pseudo registers' with the → 'hardware registers' they were allocated during register allocation (function '`alter_reg`')
- perform → 'register elimination' (function '`eliminate_regs_in_insn`')
- save and restore call-clobbered (→ 'clobber') registers around calls (main functions are '`setup_save_areas`' and '`save_call_clobbered_regs`')

References: [7], node 'Passes' at 'Reloading'.

Reload Register GCC

A hard register used by the → 'reload pass' for temporarily storing values in.

RTL GCC → 'register transfer language'

RTL Class GCC

RTL → 'expression codes' are divided into several groups, called 'classes'. References: ‘rtl.def’ at ‘GET_RTX_CLASS’; [7], node ‘RTL Classes’.

RTL Generation GCC
Uses the parse tree built up by the parser for the generation of RTL for the current function. Each operation needs certain definitions of → ‘standard names’ to be present in the → ‘machine description’. Also called → ‘expansion’. Implemented by function ‘c_expand_body’.

RTL Template MD
A part of an → ‘instruction pattern’ describing the set of acceptable RTL expressions that would match at this position. A typical example is a ‘match_operand’ expression, whose → ‘predicate’ and constraint (→ ‘constraints’) fields specify whether a ‘mem’, a ‘reg’, a ‘const_int’, or any combination of them would be acceptable at the position where it occurs in the instruction pattern. References: [7], node ‘RTL Template’.

RTS GCC Reverse topological sort order.

RTX GCC → ‘RTL’ expression. References: typedef of ‘rtx’ in ‘config.h’.

RTX Code GCC
A synonym for → ‘expression code’.

Runtime GCC
The time when a binary, which has been generated by gcc from user source files, is being run on the → ‘target machine’. See also: → ‘build-time’ and → ‘compile-time’.

sa GCC List **s**tarts with an **a**ttribute. Used in ‘c-parse.y’.

Saved Registers GCC
This refers to registers saved in either the → ‘register save area’ (callee-side) or the → ‘caller save area’ (caller-side), which stand for two separate concepts of making registers keep their value across function calls.

sc GCC **S**torage **c**lass. Used in ‘c-parse.y’.

SCSPEC GCC
See → ‘storage class’.

Sequence Point GCC
At a sequence point, all pending increment operations such as in ‘old_i = i++;’ are executed. In GCC, a sequence point is always marked by a call to ‘emit_queue’. References: discussion of the concept in [8], node ‘Warning Options’ at ‘-Wsequence-point’.

Shifting GCC
The parser shifts the next token of the input file onto its parser stacks whenever no → ‘Reduction’ is currently appropriate.

Sibling Call GCC
A call to function ‘g’ at the end of function ‘f’. Under certain circumstances, similar optimizations as with a → ‘tail call’ are possible. The implementation can be found in ‘sibcall.c’.

Signed Saturation GCC
Clipping to the maximum or minimum representable value in the case of overflow occurring during an arithmetic operation. Bytes, words, and double-words are regarded

as holding signed values. Opposite: → ‘unsigned saturation’.

SIMD Single instruction, multiple data.

SMT TI Simultaneous multi-threading technology (also known as ‘Hyperthreading’)

SP GCC Stack pointer

SPC TI Section program counter

Specs Language GCC

Specification language, used for configuring the behaviour of the compiler driver ‘`gcc`’, in particular with which options to call the compiler, the assembler and the linker. References: functions ‘`read_specs`’, ‘`do_spec`’, and struct ‘`static_specs`’.

Spilling GCC

→ ‘register spilling’

SSA GCC Static single assignment

Stack Frame GCC

A synonym for → ‘Activation Record’.

Standard Action GCC

A feature of a programming language (jump, function call, arithmetic, etc.), for which a → ‘standard name’ exists in GCC.

Standard Names GCC

Well-known names of standard operations the compiler is aware of and can employ for achieving certain kinds of results. Examples are ‘`movM`’ for moving between memory and registers as well as from registers to registers, ‘`addM3`’ for adding registers, and ‘`bCOND`’ for conditional branching. Which → ‘standard actions’ are available can be defined in the → ‘machine description’ by means of → ‘expander definitions’ and named → ‘instruction patterns’. References: Section 2.3.11 and [7] at node ‘Standard Names’.

Statement Tree GCC

The chain of statements, built up by the parser for a C function. References: functions ‘`begin_stmt_tree`’ and ‘`add_stmt`’.

Static Chain Pointer GCC

Slot in the → ‘stack frame’ pointing to the stack frame of the first ancestor function in the → ‘call chain’, which represents an enclosing scope of the current function. This is needed in the implementation of lexical scope for nested functions. [1] uses the term → ‘access link’ for this pointer and explains the concept in detail.

Static Link GCC

A synonym for → ‘static chain pointer’.

Storage Class GCC

One of the following C keywords: ‘`auto`’, ‘`extern`’, ‘`register`’, ‘`typedef`’, ‘`inline`’, ‘`__thread`’. The parser represents this as token ‘`SCSPEC`’. Note that ‘`static`’ is no ‘`SCSPEC`’ token, but a ‘`STATIC`’. References: the ‘`rid_to_yy`’ array.

Tail Call GCC

A call to function ‘`f`’ at the end of function ‘`f`’. The overhead for creating an extra

→ 'stack frame' can be optimized away doing → 'tail recursion elimination'. Sometimes, functions can be made tail-recursive by instruction reordering.

Tail Recursion Elimination GCC

An optimization in which a recursive call at the end of a function is transformed into computing the actual arguments, overwriting the current formals on the stack and then doing a simple goto to the start of the function. References:

- 'optimize_tail_recursion' $\xrightarrow{\text{calls}}$ 'tail_recursion_args'
- 'CALL_PLACEHOLDER'—gets substituted by the real call instructions as soon as a decision is made as to which calling method (sibling, tail, normal) is to be used

Target Flags GCC

Flags that can be set by means of command line options when invoking 'gcc'. This allows to support several variations of a target architecture in the same → 'machine description'. This might at some point be used to dynamically enable support for the particular features of 'C62x', 'C64x', and 'C67x', for example. References: Macro 'TARGET_SWITCHES' in [7] at node 'Run-time Target', function 'set_target_switch', and macro 'CONDITIONAL_REGISTER_USAGE'.

Target Machine GCC

The → 'machine' which 'gcc' generates Assembler and binary code for. If this is different from the → 'host machine', then the current 'gcc' binary is a → 'cross-compiler'. References: [7], node 'Configure Terms'.

Template MD

See → 'RTL template' or → 'output template'.

Template Variable GCC

A 'match_operand' or a similar expression that is used within an → 'RTL Template'. The term does not occur in the GCC documentation, but is used only within this work. References: Section 2.3.10 and [7] at node 'RTL Template'.

Temporary Slot GCC

Temporarily allocated stack location used for evaluating C expressions. Also, → 'automatic variables' are a special kind of temporary slot in GCC. References: comment at definition of 'struct temp_slot' in file 'function.c'

Thread-local Storage GCC

Variables local to threads. A GCC C-extension. [8], node 'Thread-Local'.

Thunk GCC

→ 'multiple inheritance thunk'

TI Texas Instruments

TLS GCC → 'thread-local storage'

TM GCC → 'target machine'

Toolchain GCC

The sequence of tools that is typically used in order to translate a collection of C source code files to an → 'executable', also called 'binary'. References: Section 2.2.

Trampoline GCC

A small function, automatically generated by the compiler for calls to nested functions (a GCC C-extension) via a function pointer. The trampoline is responsible for setting up the → 'static link' and redirecting to the real function. References: [7], node 'Trampolines'.

Trap GCC Instructions may 'trap' when an error condition must be signaled. Typically, some kind of signal will be raised. References: functions '`may_trap_p`' and '`may_trap_exp`'; [7], node 'Standard Names' at 'trap' and 'conditional_trap'.

Tree-Based Inlining GCC
→ 'Inlining' of C/C++ functions based on the → 'Tree Representation' of the source program. References: 'tree-inline.c' and [7], node 'Passes'.

Tree Code GCC
Tells the kind of a node in the parse tree. Macro '`TREE_CODE(`⟨*node*⟩`)`' can be used to access it. Tree codes are defined in 'tree.def' (language independent) and 'c-common.def' (C and C++).

Tree List GCC
A data structure which is a chain of nodes of type '`union tree_node`', accessed as '`struct tree_list`' via macros '`TREE_PURPOSE`' and '`TREE_VALUE`'. The next link of the chain can be accessed by macro '`TREE_CHAIN`'. References: [7], node 'Trees'.

Tree Optimization GCC
Optimization of the → 'Tree Representation'. Mainly, → 'Tree-Based Inlining' is performed.

Tree Representation GCC
The internal representation used by GCC to represent C and C++ source programs. References: [7], node 'Trees'.

ts GCC → '**T**ype **s**pecifier'. This abbreviation is used in 'c-parse.y'.

TS GCC Topological sort order

Tying GCC The local → 'register allocator' (works on 'basic blocks') "ties" two registers A and B together, if A dies by being copied to B. A register is said to "die" in an instruction, if it is the last to use it in the basic block. Tied registers receive the same → 'quantity number'. References: comment at top of file 'local-alloc.c'.

Typequal GCC
See → 'type qualifier'.

Type Qualifi er GCC
One of the keywords '`const`', '`volatile`', or '`restrict`'. The parser represents this as token '`TYPE_QUAL`'. References: macro '`TYPE_QUALS`', function '`build_qualified_type`'.

TYPESPEC GCC
See → 'Type Specifier'.

Type Specifi er GCC
One of the following elementary C types '`int`', '`char`', '`float`', '`double`', or '`void`' optionally combined with modifiers like '`signed`', '`unsigned`', '`short`', '`long`', and '`complex`'. The parser represents this as token '`TYPESPEC`'.

Unit GCC A synonym for → 'byte', in the context of → 'RTL'.

Unsigned Saturation GCC
Clipping to the maximum or minimum representable value in the case of overflow occurring during an arithmetic operation. Bytes, words, and double-words are regarded as holding unsigned values. Opposite: → 'signed saturation'.

valid GCC An operand is valid for mode M if it has mode M, or if it is a '`const_int`' or '`const_double`' and M is a mode of class '`MODE_INT`'. References: [7], node 'Arithmetic'.

varray GCC
→ 'virtual array'

Virtual Array GCC
Data structure in GCC. Dynamically growable array for various element types. Can also be used like a stack. → 'Object stacks' are an alternative data structure. References: 'varray.h' (and 'varray.c')

Virtual Register GCC
Serves as address to a location on the stack, for which an absolute address cannot be computed immediately, because other items put later on the stack might cause it to change as the compilation progresses. One way for function '`instantiate_virtual_regs`' to eliminate virtual registers is to replace them by '`plus`' expressions, each consisting of a reference to a hard register and a constant offset. Another way is to allocate a new → 'pseudo register' and emit instructions to compute the sum. References: Section 2.3.1 on page 11; [7], node 'Regs and Memory' at '`VIRTUAL_`$*$'.

VLIW Very large instruction word

volatile GCC
A → 'type qualifier' for variables which should not be put into a register, as that would cause them to have an undefined value after a '`longjmp`'. There is a GCC extension, defining a '`volatile`' function as one that never returns. This extension is deprecated, however. References: [7], node 'Interface'.

wfl GCC 'With file location'. References: → 'tree code' '`EXPR_WITH_FILE_LOCATION`'; function '`build_expr_wfl`'.

yylsp GCC The variable used by bison for the **l**ocation **s**tack **p**ointer. For example, used for printing line numbers in error messages and for generating debugging information.

yyssp GCC
The variable used by bison for the **s**tate **s**tack **p**ointer. The state stack is used for keeping track of which tokens and non-terminals have been seen so far and which reduction rules may therefore be applied.

yyvsp GCC
The variable used by bison for the **v**alue **s**tack **p**ointer. The value stack stores semantic values of tokens and non-terminal symbols.

References

[1] **Compilers—Principles, Techniques, and Tools**. Alfred V. Aho, Ravi Sethi, and Jeffrey D. Ullman. Addison-Wesley, 1986, ISBN: 0-201-10194-7.

[2] **The Annotated C++ Reference Manual**. Margaret A. Ellis and Bjarne Stroustrup. Addison-Wesley, 1986, ISBN: 0-201-51459-1.

[3] **The Free Software Definition**. Free Software Foundation. http://www.gnu.org/philosophy/free-sw.html.

[4] **Compile C Faster on Linux**. Christopher W. Fraser and David R. Hanson. http://www.cs.princeton.edu/software/lcc/doc/linux.html.

[5] **Host/Target specific installation notes for GCC**. Free Software Foundation. http://gcc.gnu.org/install/specific.html.

[6] **GNU Binutils**. Free Software Foundation. 2000–2004. http://gcc.gnu.org/gcc-3.3/.

[7] **GCC Internals Manual (GCC v3.3)**. Free Software Foundation. May 2003.

[8] **GCC Manual (GCC v3.3)**. Free Software Foundation. May 2003.

[9] **GNU Bison**. Free Software Foundation. May 2003. http://www.gnu.org/software/bison/bison.html.

[10] **GNU Bison Manual**. Free Software Foundation. May 2003. http://www.gnu.org/software/bison/manual/.

[11] **GNU Compiler Collection v3.3**. Free Software Foundation. May 2003. http://gcc.gnu.org/gcc-3.3/.

[12] **Watcom C/C++ and Fortran**. SciTech Software. 2004. http://www.openwatcom.org/index.html.

[13] **TMS320C6000 Assembly Language Tools User's Guide (spru186)**. Texas Instruments. http://www.ti.com/.

[14] **TMS320C6000 CPU and Instruction Set Reference Guide (spru189)**. Texas Instruments. http://www.ti.com/.

[15] **TMS320C6000 Optimizing Compiler User's Guide (spru187)**. Texas Instruments. http://www.ti.com/.

[16] **TenDRA**. The TenDRA Project. 2001–2004. http://www.tendra.org/.