Porting The Gnu C Compiler to the CRIS architecture

Hans-Peter Nilsson*

September 8, 1998

Abstract

This master's thesis report presents the work of retargeting the Gnu *C* compiler (Gnu CC, GCC) to a processor with a fairly simple general architecture; the CRIS, resulting in the *gcc-cris* compiler. The report is aimed at describing the work of porting, in such a manner as to be usable as a guide to porting GCC for use as a cross-compiler for similar architectures.

The target system type is an embedded system, centered around networked communication applications.

In aiming a target system, GCC has a rich set of parameters, and a special machine description language. Many of the machine parameters are analyzed and explained. The version originally ported was GCC version 2.1, but the report is up-to-date to at least version 2.8.1 and to the experimental *egcs* variant up to at least version 1.1.

The target system and the thoughts behind it are described. Measurements on performance of actual applications, while changing key machine parameters of GCC, are presented.

As of September 8, 1998, the compiler used in production is based on GCC version 2.7.2, with adjustments for version 2.8.1 in progress.

*Email: <hp@axis.se>. Advisors: Per Zander, Axis Communications AB. Per Andersson, NetCore (former of Department of Information Technology, Lund Institute of Technology). Mats Brorsson, Department of Information Technology, Lund Institute of Technology.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 Background

1.1.1 Reason for this work

In the networked communications-appliance business, the need to cut costs together with the need for special communication circuits, bring on the demand to integrate as many functions as possible into application-specific circuits, ASIC:s. The complexity at the "hardware level" of current communication protocols and other interfaces, makes the CPU of an embedded system one of the simplest components, while still having a relatively high cost. The wish to integrate the CPU into the ASIC comes forward. This also gives the possibility of simplified and faster access to communication functions.

Practical and business restrictions, such as per-part royalty-fees and the gate-size and technology of offered ASIC-block CPU implementations, limits the choice of popular CPU:s with necessary performance. Consider the relative simplicity of a basic CPU block (without optimizing features such as speculative execution, caching or branch-prediction), compared to the rest of e.g. a network communications-device ASIC. Then, if just for the sake of hardware design complexity, designing a streamlined home-brew architecture becomes realistic.

Because of portability considerations and complexity of higher-level protocols, most of the code for such an embedded system is written in a portable language with necessary structure, where *C* is often chosen. Thus, to make the home-brew architecture usable, we need — among other tools — a *C* compiler for it.

1.1.2 Compilers

A compiler is normally specific for the target CPU and system on which the compiled code will run. It is also naturally language-dependent. However, large parts of the code for a compiler are actually neither programming-language- nor target-system-dependent, and so can be re-used for other CPU's, systems and programming languages, if written properly.

The compiler is a very complex program. As such it is prone to limitations and errors.[1] Development time and cost therefore rules out writing a compiler from scratch, for most corporations. Availability of source code for the compiler is of high priority, so possible bugs can be tracked down and corrected immediately. The alternative is the less structured and uncertain way of rewriting the application code to work around them.

When searching for different available retargetable compilers to compare, at the preparation phase of this work, very few were found besides GCC.[2] They all had a very limited set of targets, making it probable that they were hard to re-target, and many were more or less toy implementations of the *C* language — important features such as structures or arrays were not implemented. The closest "competitor" was LCC,[3] but it too had few users, few targets and in general had inferior capabilities compared to GCC. However, GCC was fairly robust; being widespread and constantly supported (albeit to a great extent by volunteers with unknown priorities), and it has been ported to more targets than any other *C*-compiler. Also, it has been used in production for many years, and is therefore probably sufficiently stable. Bug-reports are dealt with swiftly, and there are newsgroups and mailing-lists dedicated to GCC. Known bugs are documented and bug-lists are publicly available, see appendix C. See appendix B for information on how to get GCC.

If all involved parts are modifiable, then it is easier to adjust a target for the demands and features of an existing compiler than to adjust the peculiarities of the compiler to a specific target. The architecture was not finished at the time of this work, so some properties of the intrinsics of GCC were taken in consideration at the final target architecture development, after GCC was selected. Most notably this affected the number of registers and the valid addressing-modes of some rare instructions.

1.2 This work

1.2.1 Goal

The goal for this work was to produce a compiler based on GCC, to be used with a home-brew architecture, called *CRIS*, short for Code Reduced Instruction Set.[4] The intended result was limited to the main compiler part; that which translates from C into assembler code in (ASCII) text format.[5]

[1] The size of the *C* code for GCC for the *C* language only, is about 300k ... 400k lines, depending on how much is auto-generated by the target description.

[2] This still holds, maybe due to the widespread availability of GCC for most architectures. The effort that must be put into building a compiler from scratch, for a system that is not in widespread use, does not pay off in the presence of the GCC compiler.

[3] For information on LCC and other compilers, see appendix C and the [LCC] book.

[4] At that time theoretical, although later implemented in the ETRAX chip.

[5] As will be seen, for practical reasons this restriction could not be enforced.

1.2.2 Porting

The work of porting a compiler includes two major parts:

ABI specification

Most often the ABI[6] is fixed by previous work that you have to be compatible with, for that architecture. Basically, the ABI is the fundamentals of your machine, like how *C* types map to bytes and bits, memory layout of the types, how to call functions and how to return values.

This is needed for those instances where there is a pre-compiled library, with no source code available, that has to be linked and included in a project. Most often the case is with system libraries. Other times you have to call program parts that are written in other languages, and there has to be a documented interface for how to call (and be called from) assembler code.

The machine description

This is the work of describing the architecture and the ABI to the compiler. In GCC this is done through *C* files and a special machine-description language.

1.2.3 The result

The work described in this report was originally done for GCC version 2.1. The result, the gcc-cris compiler, has been in constant use since then, and has later on been updated up to version 2.7.2 as of this writing. The same port is used in production for code written in *C++* using the front end support in GCC for that language.

1.2.4 This report

No language-specific details on parsing, theory or techniques for writing compilers are investigated here. The focus is on porting by adding a machine description.

The target system is described in chapter 2. GCC, its porting mechanisms and philosophy are presented in general in chapter 3. The porting and thoughts behind it are found in chapter 4. In chapter 5, some measurements of performance are presented. Conclusions are found in chapter 6, and some tricks and limitations of GCC are discussed in chapter 6.4. Please confer to appendix D for definitions of the used terminology.

Although this report primarily describes the original porting, important differences or facts that have been discovered since then have been merged to increase usability. Footnotes have been used when appropriate.

This report can be found in various formats at
<URL:ftp://ftp.axis.se/pub/axis/tools/cris/misc/> under the names

[6] See appendix D.

`rapport.`*`format`*, for example `rapport.pdf`. An errata will be located at <URL:ftp://ftp.axis.se/pub/axis/tools/cris/misc/report-errata.html>.

Suggestions for reading

A reader with no previous knowledge of GCC or the CRIS architecture can read this report straightforward for best apprehension. Any reader with previous knowledge of the CRIS architecture and its usage in gcc-cris, can skip chapter 2 and read the rest. If the reader is oriented in GCC and porting to different architectures, then chapter 3 can certainly be ignored. Any other readers should probably skip chapter 3.2.1 at the first reading.

For optimal apprehension of the technical contents, the reader should be generally oriented in computer science, with at least an introductory-level course on compilers and computer architecture. Knowledge of the *C* programming language is assumed. It is useful to have general knowledge of common microprocessor architectures, such as the families of Intel i386, Motorola MC68K, DEC VAX and National Semiconductor NS32K. It is recommended to read [Stal 92] at the time of reading this report, but not a precondition.

Chapter 2

The target system

2.1 The target architecture

GCC is specifically aimed at CPU:s with several 32-bit general registers and byte-addressable memory. Deviations from this is possible. In short, you can make a port of GCC for a target with 16-bit registers, but not a decent implementation for a processor with only one general register.[1] Also, the size of the memory must not be bigger than what can be addressed from a register. For example, you should avoid segmented-memory architectures. The CRIS architecture was designed with this in mind.

2.1.1 Registers

The registers are general 32-bit registers, named `r0` ... `r15`. They are interchangeable in function, except for `r15` which is actually the program-counter, `pc`. There are also 16 special registers, but only a few are applicable to GCC. One of them is the condition-code register, `ccr`, implicitly used by GCC. Another is the subroutine-return pointer, `srp`, used implicitly by the `jsr` and `ret` instructions.

There is no hardwired stack-pointer in the CRIS architecture. However, `r14` is most convenient for this purpose,[2] and is therefore treated as the necessary stack-pointer. It will be referred to as `sp` in the following. I will also use the notation *rX*, *rY* or *rZ* when referring to up to three (possibly, but not necessarily) different registers.

[1] Also known as "accumulator" architectures.

[2] Mostly because all other normal registers can then be moved to and from stack by the `movem` instruction, when saving and restoring register contents at the prologue and epilogue of a function.

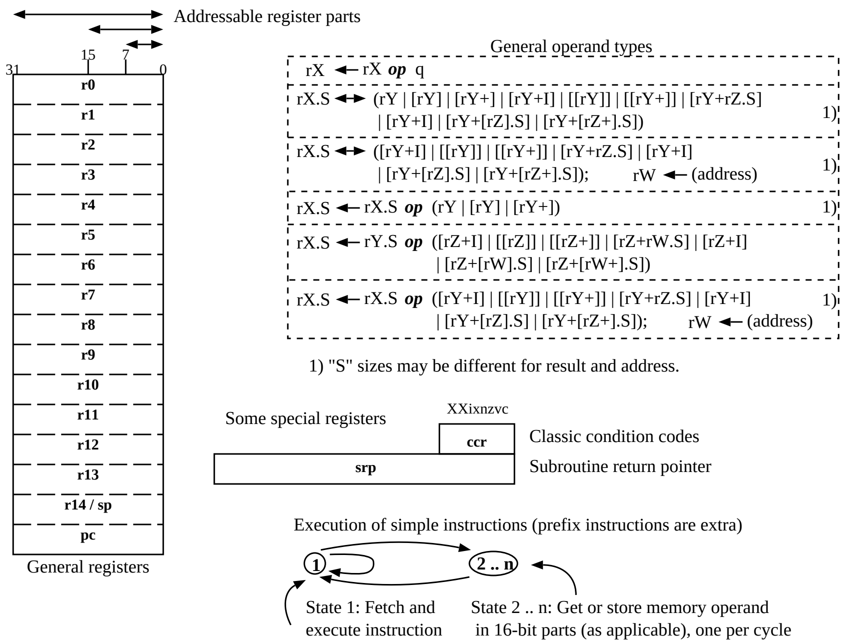

Figure 2.1: CRIS architecture: overview of parts relevant to GCC

2.1.2 Sizes

Operand sizes are byte, word, and dword, symbolized by b, w and d respectively, and with (capital) S as any one of that set. Operations on word or byte do not change the more significant parts of the register. For arithmetic on larger data than dword, a special "extend" flag is used for easy carry-passing.

2.1.3 Addressing modes

Most "binary operator" instructions are *two-operand* instructions, with one register the same as the destination and one of the source operands. The other operand can be in memory, be a constant or be another register. The basic addressing-modes are very few: quick immediate, register, indirect register and indirect register with post-increment, symbolized by q, rX, *[rX]* and *[rX+]* respectively. An operand with a basic addressing-mode except q, is denoted with (lower-case) s. More complex addressing-modes are implemented by special instructions called *address prefix instructions*, that modify an s memory operand. They are called *prefixed addressing-modes*.

The following addressing-modes, interpreted by the assembler, are implemented through address prefix instructions:

Mode description	Assembler syntax	Comment
Indirect with offset	[rX+I]	I is a constant $-128 \leq I \leq 127$[3]
	[rX+[rY].S]	
	[rX+[rY+].S]	The S is the size of the sign-extended access and increment
Double indirect	[[rX]]	
	[[rX+]]	
Indirect plus scaled	[rX+rY.S]	rY is multiplied by sizeof(S) (i.e. 1, 2 or 4)

Double indirect is also available in a version with post-increment of rX, mostly used implicitly by the assembler when rX is `pc`. This results in addresses of the form `[`*address*`]`, where *address* is a constant. This is often used for reading and writing a single global variable in *C*.

Immediate constants other than what fits in the six bits of a quick immediate constant, are expressed using indirect post-increment on `pc`. As a special case, post-increment on `pc` for byte operands will cause an increment by two, to keep `pc` word-aligned.

2.1.4 Instructions

An opcode is always 16 bits long, counting address prefixes as separate instructions. For each arithmetic operation, both the signed and the unsigned condition results is stored in `ccr` like with e.g. the Motorola MC68K series.

All logical, arithmetic and move-into-register instructions update the condition codes, except for `addi` and the side-effect-part of the prefixed addressing-modes with side-effects (see page 16). Moves into memory do not update the condition code register. Most condition-code-results reflect useful conditions, in that the condition reflects a compare by zero with the result of the operation.

The opcode table is seen below. The CRIS architecture changed somewhat from the time of the porting to when the first hardware was manufactured, as marked by the footnotes. Only instructions that are recognizable by GCC are included. Any mentioned operator is the same as that *C* operator.

Mnemonic	Operands, left to right	Sizes[4]	Description
move.S	s to r, r to s		
moveq	-32 ... 31 to r	d	
movs.S	s to r	b w	Sign-extend s to dword r.

[3]The assembler transforms $I < -128$, $I > 127$ and symbolic (unknown) I into the [rX+[rY+].S] form, using `pc` as rY and word and dword as S. The dword mode was a late addition before CRIS went silicon, to be used for symbolic expressions unknown at compile time. Note that the specific byte form of this instruction, [rX+[rY+].b] (with and without post-increment), is a different address prefix instruction than the [rX+I], where I is known to be $-128 \leq I \leq 127$.

[4]An empty field means that a S stands for any of *b*, *w*, or *d* for the source operand.

Mnemonic	Operands, left to right	Sizes	Description
movu.S	s to r	b w	Zero-extend s to dword r.
clear.S	s		Set s to all zeroes.
movem	s to r, r to s	d	Move all registers `r0` ... r inclusive, to or from memory pointed to by s.
link[5]	rX to `[`rY`]`	d	Move rX to `[`rY`]`, then rY to rX.
abs	rX to rY	d	Store the absolute value of rX into rY. No action for the value 0x80000000.
add.S	s to r		
addq	0 ... 63 to r	d	
adds.S	s to r	b w	Add sign-extended s to dword r.
addu.S	s to r	b w	Add zero-extended s to dword r.
addi	$rX.S$ to rY	d	Add scaled rX to rY. The scale factor is 1, 2 or 4 as determined by S.[6]
sub.S	s from r		
subq	0 ... 63 from r	d	
subs.S	s from r	b w	Subtract sign-extended s from dword r.
subu.S	s from r	b w	Subtract zero-extended s from dword r.
neg.S	rX to rY		$-rX$ stored in rY. No action for the value 0x80000000.
test.S	s		Compare s to zero.
cmp.S	s to r		Sets flags as if the operation was a subtraction of s from r.
cmpq	-32 ... 31 to r	d	
cmps.S[7]	s to r	b w	Compare dword r with sign-extended operand.
cmpu.S[7]	s to r	b w	Compare dword r with zero-extended operand.
mul[8]	rX by rY	d	Lowest 32 bits of rX * rY are stored in rY. [9]

[5]This instruction was removed before CRIS became silicon.

[6]This instruction does not affect the condition-code flags.

[7]These instructions were added (on the authors suggestion), before CRIS became silicon.

[8]This instruction was removed before CRIS went into silicon. It was replaced with an instruction called *mstep*, which implements a part of the multiplication; shift-and-conditionally-add. The exact details are stated in [Axis 95].

[9]The "bit patterns" of the result are the same for signed and unsigned multiply (unless

Mnemonic	Operands, left to right	Sizes	Description
div[10]	rX, rY		Unsigned divide and modulus: rX / rY is stored in rY; rX % rY is stored in rX.
abs	rX to rY	d	
and.S	s to r		
andq	-32 ... 31 to r	d	
or.S	s to r		
orq	-32 ... 31 to r	d	
xor	rX to rY	d	rX ^ rY stored in rY.
not	rX to rY[11]	d	~rX is stored into rY.
lsr.S	rX by rY		Logical right-shift: unsigned $rX \gg rY$ is stored in rX.[12]
lsrq	rX by 0 ... 31	d	
lsl.S	rX by rY		Logical left-shift: $rX \ll rY$ is stored in rX.[12]
lslq	rX by 0 ... 31	d	
asr.S	rX by rY		Arithmetic right-shift: signed $rX \gg rY$[13] is stored in rX (the sign-bit is propagated).
asrq	rX by 0 ... 31	d	
btst	bit rX of rY	d	Set negative-flag if the bit with number rX in operand rY is set, and set the zero-flag if the all bits from bit 0 and up to the bit number rX in operand rY are zero.
btstq	bit 0 ... 31 of rX	d	
bCC	*8- or 16-bit offset*		Branch on condition code CC (including branch-always). The offset is $-254 \leq pc \leq 256$ bytes for an 8-bit offset, and $-32764 \leq pc \leq 32770$ bytes for a 16-bit offset, counting from the origin of the bCC instruction.

overflow occurs), so there's only need for one multiply-instruction.

[10]This instruction was removed before CRIS went into silicon. It was replaced with an instruction called *dstep*, which implements a part of the division; shift-and-conditionally-subtract. The exact details are stated in [Axis 95].

[11]The addressing-mode of this instruction was modified before CRIS went into silicon; the destination register must now be the same as the source register.

[12]The count is indicated in the six (not five) least significant bits. The result is 0 if the sixth bit is set.

[13]See note [12], except that the result is -1 if both the sign-bit and the sixth bit were set.

Mnemonic	Operands, left to right	Sizes	Description
jsr	s		Store `pc` into `srp`, load address s into `pc`.
ret			Store `srp` into `pc`.
bound.S	r by s		Set r to unsigned minimum of r and zero-extended (as necessary, to dword) s.
sCC	r	d	Set r to 1 if the specified condition code is true, or 0 if false.
nop			

There are some differences in the CRIS architecture, compared to popular microprocessor architectures:

- The `pc` is not saved on the stack at a subroutine call (the `jsr` instruction). Rather, it is placed in `srp` and the called subroutine must take care of saving this register when needed.
- The branch instructions all have a one-instruction delay before they are executed, a *delay-slot*.[14] The delay-slot instruction is located at the address after the branch, and will be executed before the branch is taken. It must fit completely in 16 bits, so no prefixed addressing-mode is possible. It must not use `pc` in its operands. If no useful instruction can be scheduled for the delay-slot, a `nop` instruction has to be placed there.
- The prefixed addressing-modes have an optional assignment side-effect feature.[15] This means that you can optionally assign the value of the effective address to a register. In assembler notation, this is for example [rY=rX+I] and [rX=rY+$rZ.S$]. Nothing is won if rX and rY are the same and if only registers are involved; then you could just as well use a separate `addi` instruction. The [rX=rX+I] case still gives one word shorter code for the case where $-128 \leq I \leq -64$ or $64 \leq I \leq 127$, since the 8-bit signed offset is contained within the address prefix, and not in an extra word as it would be with the corresponding *add* or *sub* instruction.
- When using the prefixed addressing-modes, you can specify three-operand behavior[15] for the instruction. Instead of having one register as both being a source and the destination, you could perform the operation on the register and the specified memory location, and store the result in another register. Example: `adds.b [r0+42],r1,r2` would take the byte at address `r0` plus 42, sign-extend it to dword, add it to `r1` and store the result in `r2`.

[14] Before CRIS became silicon, the *ret* instruction was modified to have a delay-slot too.
[15] Note that only one of these options is available for each instruction.

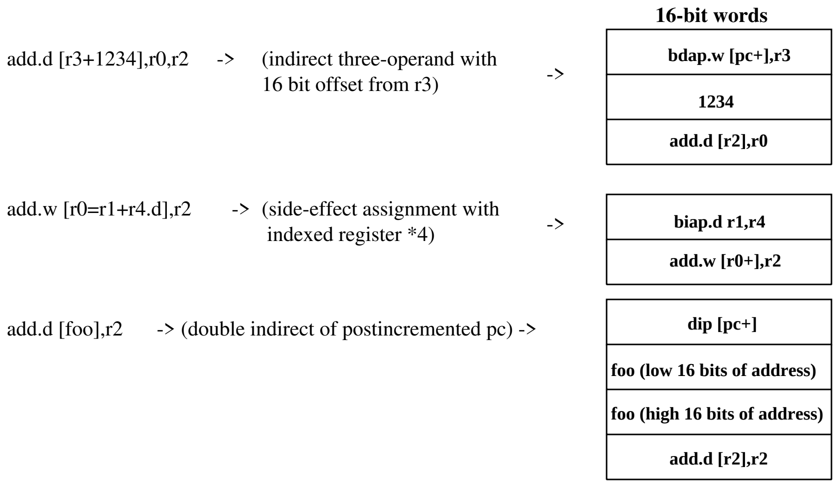

Figure 2.2: Prefix instructions: placement and assembler expansion

The prefix instructions *can* be expressed explicitly, but in practice they are never used that way, and will not be described further.

Since none, or very few floating point operations are needed in the typical intended application, they are not implemented in hardware.

In order to keep opcodes as compact as possible, some reductions in orthogonality of the valid addressing-modes had to be performed. Otherwise, the various addressing-modes for rarely used instructions would take up room that could be better used for other instructions. Shift operations must have their count in a register, or expressed as a quick immediate value. Also, because of their relative rareness, the instructions `abs`, `neg`, `not` and `xor` are even more restricted:

abs only works for registers, and only in dword mode.

neg only works for registers.

not only works for one register, being both source and destination, and only in dword mode.[16]

xor only works for registers, and only in dword mode.[16]

To support the *C* `switch { case ...: }` construct effectively, a special instruction called `bound` was included. Its result is the unsigned minimum of the operands as the result. See chapter 4.3.4 for a discussion on how GCC uses this.

[16]This is still usable for smaller modes as well, if a change in the more significant bits can be ignored.

2.2 The target run-time system and libraries

No run-time system or libraries existed for the CRIS architecture at the beginning of this work. After probing around Internet[17]-related resources and in-house resources for available libraries, I was able to get together a working system library as described in [ANSI C], together with a floating point library. However, the description of that work is out of scope for this report.

A necessary run-time library most often already exists for a given system or can be easily modified from that of a similar system or retrieved from a free source. See appendix A for resources.

GCC itself contains a halfway-finished IEEE floating point library, *floatlib*. With small modifications I made it work for the simple calculations that were needed in the intended application.[18] Absolutely correct rounding and correct error handling at the bounds of the representation as specified in the IEEE specification was not of interest.[19]

2.3 Simulation environment

At the beginning of this project, there already was a simulator, mainly for the purpose of testing the assembler. I modified it during this project, mostly by adding hooks into the host file system so actual file operations could be performed, and memory allocation hooks for heap allocation and automatic stack allocation. Other small modifications were made, to make it report statistics for instructions and addressing-modes. The instruction statistics were interesting during the development of the port and for trimming the architecture, but for the compiler tests, the primary output was the total number of cycles.

[17] Which was not a fraction as resourceful in 1992 ... 1993 as it is now.

[18] See chapter 4.6.1

[19] A newer "bundled" floating-point library, called `fp-bit`, appeared with GCC version 2.6. Initial tests show that it is smaller but slower. It is used for many of the ports and should be a better choice in the long run.

Chapter 3

The GNU *C* compiler

First of all, what is said in this chapter is almost completely deducible from what can be read in [Stal 92], except that you will probably have to read the latter a few times to get an overview.

3.1 The compiler system

People often have vague and over-extensive views of what is in a compiler, probably because of the wide-spread integrated development environments, where the parts are not visible.

If you imagine your development environment as parts, you might realize that the compiler really may be not more than a program that translates from *C* into assembler code and nothing else. Well, there is the notation of a *C* preprocessor; a part that expands the *preprocessing directives*. Just as the other compiler parts it can be a separate program, one that feeds the main compiler program.

This division of the system, in this case the compiler system, into parts is natural on a system of Unix-flavor, which is the birthplace and natural habitat of GCC.

As GCC is intended to be only one part of a compiler system, it comes with just enough facilities to create assembler code from *C*, and very little more.[1] Not even *standard header-files*[2] to a *standard library*. Specifically, neither the standard library nor the assembler program are included. They are all separate programs and files, just used together at the moment of the compilation.

On the other hand, GCC contains generic code-generating machinery that connects to other programming language *front ends*, giving access to multiple other languages, once there is a port of GCC to a target system.[3]

[1] Unless of course, you fetch the other parts, normally available at the same resource.
[2] Although some machine-specific header files are generated when the compiler is built.
[3] At this writing, front ends for *Fortran*, *C++*, *Ada* and *Pascal* exist. See Appendix A.

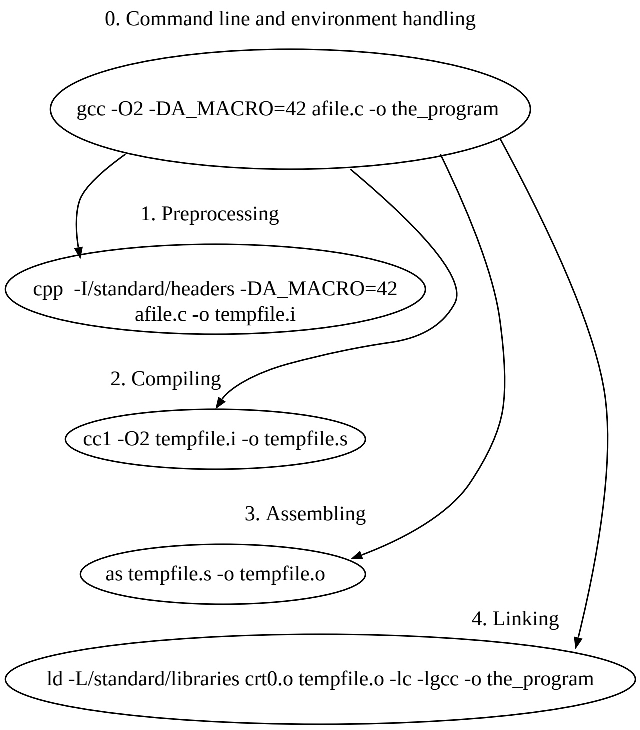

Figure 3.1: Compiler parts and calling sequence

3.1.1 The compiler parts

When GCC is used, it is commonly called as in one of the following examples:

1. `gcc` *-switches one_or_more_sourcefiles*`.c -o` *program*
2. `gcc` *-switches* `-c` *sourcefile*`.c`
3. `gcc` *-switches objectfile1*`.o` ... *objectfileN*`.o -o` *program*

The first alternative is used when compiling one or very few source files "directly" into a program. The second alternative is used when compiling one of several source files for a project into an object file. The third alternative is used after several runs of the second alternative, when linking together the object files into a program. More alternatives exist, but are seldom used and not of interest here.

All three alternatives actually look the same to the compiler parts. The program executed under the name of `gcc` is just a front that handles options and temporary files and calls the real compiler parts:

cpp The *C* preprocessor. It takes care of preprocessor directives, such as include-files and macro expansion. It also removes comments. The result is a file with the *C* code, lots of whitespace and some line-numbering directives, that the compiler core can use in warning and error messages. This program comes as part of GCC.

cc1 The compiler core, a.k.a. the compiler proper. It includes both the language *front end* and the target machine *back end*. This program comes as part of GCC.

as The assembler.

ld The linker.

These programs are always called by these names by the `gcc` shell. If any particular system has another name, a *symbolic link* or shell program with the expected name is installed (in a GCC-specific directory, so the impact on the system is minimal) when GCC is installed.

The `gcc` program keeps track of where to find the system header files (the ones that are included using brackets, like in `#include <stdio.h>`). Actually, it is the only compiler part that needs to keep track of where specific directories and files are located.

Referring to the numbered examples of `gcc` calls above: For the first and the second alternatives, it calls the *C* preprocessor `cpp`. It passes along the location of system header-files and some of the command-line arguments to `cpp`, to specify the input to be from one *C* source file and that the output, the *preprocessed* C *code* goes to a temporary file. After `cpp` is finished, `gcc` calls `cc1`, specifying the file where the input is, and where to put the assembler code; normally another temporary file. When `cc1` is finished, the same thing happens for the assembler. For alternative two, the object code ends up, not in a temporary file, but in a file with the same name as the source file, only suffixed with `.o` instead of `.c`. This sequence is repeated for all *C* source files. At last, after the assembler is done, `gcc` calls the linker `ld` (for the first or the third alternative), specifying all object files, the location of any libraries, and where to put the linked program.

The focus of this report is on the main compiler part, `cc1`, as part of a cross-compiler.

3.2 The porting mechanisms

When porting GCC to a system, the system, especially the processor and its immediate machinery, has to be described in detail. This is done partly with *C* macros, stating values of general properties, and partly by detailing the instructions and their operand types using a special language.

The descriptions are located in three major files: a *C* macro file, canonically called `tm.h`, a *C* file called `tm.c`, and the machine description file, `md`. In the GCC source code, they are called *`target`*`.h`, *`target`*`.c` and *`target`*`.md`, but the canonical names are used by the other source files, after the compiler is configured.[4]

tm.h	C macros for machine fundamentals, compiler environment, machine description support and application binary interface	`#define TARGET_SWITCHES ...`
		`...`
		`#define BITS_PER_WORD ...`
		`...`
		`#define CONST_OK_FOR_LETTER_P(VALUE, C) ...`
		`...`
		`#define FUNCTION_PROLOGUE(FILE, SIZE) ...`

tm.c	C functions from macro expansion, machine description support	`void function_prologue(file, size)`
		`{ ... }`
		
		`int orthogonal_operator(x, mode)`
		`{ ... }`
		
		`int simple_epilogue()`
		`{ ...}`

md	Machine description	`(define_attr "slottable" "no,yes,branch"`
		` (const_string "no"))`
		`...`
		`(define_insn "addsi3"`
		` [(set (match_operand:SI 0 "register_operand" "...") (plus...)] ...)`
		`...`
		`(define_peephole`
		`  [(set (...) (...))] ...)`

Figure 3.2: Source parts of the compiler back end

There is also a file describing the host environment called `xm.h`, but it is not described here. After all, moving the compiler to an already-natively-ported-to different host system does not pose any new problems. Some other files are sometimes needed, like parts for the `Makefile` used when building the compiler, but they are not of specific interest here. The target-specific parts of the variable-arguments support files <`varargs.h`> and <`stdarg.h`> will be described in a section of their own.

Throughout the GCC documentation, the term *mode* is used to describe both the size and type of an operand. For example, for a byte-addressable machine with 32-bit registers, the term `QImode` is used for a *byte*, `HImode` for a *word*, and `SImode` is used for a *dword*. Floating point modes are `SFmode` and `DFmode` for single and double precision, often mapped to IEEE-754 32-bit and 64-bit sizes.

3.2.1 The *C* macros: `tm.h`

Machine and ABI properties are described using *C* macros. In the following description, it is inevitable to duplicate information available in [Stal 92], subsections of section *Target Macros*. To minimize this, while still keeping this report self-contained, I will be as brief as possible. Not all necessary macros

[4] See page 51.

are included; sometimes GCC has no default when there should have been an obvious definition. Sometimes a macro is rarely used in machine descriptions, or has a suitable default. Do not use the following description as anything else than an introduction. The macros are presented in tabular form, omitting most of the macros that are not applicable for ANSI *C* and CRIS. The following order of categories is followed, with the order of [Stal 92] coming second:

Compiler environment For example, assembler syntax, how and what switches to pass between compiler parts and where to find header files and system libraries.

Fundamental machine properties Like big or little endianness, accessible sizes, number of registers, register types and addressing-modes.

ABI How functions are called, and related topics. Details you would be interested in, if you would call a C function from assembler code, or vice versa.

Machine description support Definitions that are directly used in the `md` machine description, or to output the results from internal representation.

Most of the macros are supposed to be defined as an expression giving zero or non-zero, while some should just be defined or left undefined to state a property. If the macro must *not* have a *C*-expression definition, or if it might be left undefined, this is stated in the table.

For macros with similar use and names, the combinable parts have been listed inside { } symbols. For example, {TEXT, DATA}_SECTION_ASM_OP denotes TEXT_SECTION_ASM_OP and DATA_SECTION_ASM_OP.

Macro category Macro name(s)	*Node/Section in [Stal 92]* Comment
Compiler environment	*Driver*
SWITCH_TAKES_ARG (CHAR) ... INCLUDE_DEFAULTS	Various macros for environment definitions, such as basic switch-passing and the location of system files. All of these macros will default to usable values, if you're running the compiler on a Unix-type system. Note that the include-path for the header files will *not* include the header files for the host system, when a cross-compiler is built.

continued on next page

continued from previous page

Macro category Macro name(s)	*Node/Section in [Stal 92]* Comment
	Run-time Target
CPP_PREDEFINES	Describes what of system- and architecture- specific macros should be present. If present, the value must be a space-delimited constant `char *`.
TARGET_SWITCHES	This definition is an array of structures for on/off-type options passed to the compiler, prefixed with `-m`. These options are used to generate code for different architecture versions, slightly different ABI:s, or just testing something new.
TARGET_...	The names of the macros for testing the compiler options defined in TARGET_SWITCHES, should by convention have this prefix. The macro definitions should be expressions testing the variable `target_flags`.
TARGET_VERSION	If present, it should be defined to e.g. `fprintf (stderr, "TARGET V1.0");` giving the target-specific output you want with `gcc -v`.
	Type Layout
TARGET_BELL ... TARGET_CR	Defines the values for \a ... \r (canonically ASCII `bel` ... `cr`). No defaults, unfortunately.
	Library Calls
{MUL, DIV, UDIV, MOD, UMOD}{DI, SI}3_LIBCALL	The name of the corresponding library routine performing the specified integer operation for the specified size. Not needed if there is a specific instruction performing the operation. There are default names corresponding to the standard pattern names, e.g. `__mulsi3`.

continued on next page

continued from previous page

Macro category Macro name(s)	*Node/Section in [Stal 92]* Comment
TARGET_MEM_FUNCTIONS	Defined if calls to the `memcpy()` and `memset()` (ANSI and System V) functions should be generated when needed. Otherwise, the BSD functions `bcopy()` and `bzero()` are used.
	Sections
{TEXT, DATA}_SECTION_ASM_OP	For the GNU assembler, these macros should have the definitions `".text" and ".data"` respectively.
	File Framework
ASM_FILE_START (STREAM)	*C* code to output to STREAM necessary prologue text to the assembler. If the assembler has a mode where it can skip some of the input processing, if the input follows some rules, it might be a good idea to specify that mode here, and make sure that GCC outputs assembler code that follows those rules.
ASM_FILE_END (STREAM)	Analogous to ASM_FILE_START.
ASM_APP_ON	A string to be output before text from `asm`-directives, for example to revert the assembler-input-mode introduced with ASM_FILE_START.
ASM_APP_OFF	A string. Analogous to ASM_APP_ON; probably the same string as in ASM_FILE_START.
	Data Output
ASM_OUTPUT_{DOUBLE, FLOAT, INT, SHORT, CHAR, BYTE } (STREAM, VALUE)	*C* statements to output data of that specific type to the assembler.
ASM_OUTPUT_ASCII (STREAM, PTR, LEN)	*C* statement to output a “string” of ASCII characters.
ASM_OPEN_PAREN, ASM_CLOSE_PAREN	How to group arithmetic expressions for the assembler. Most use `"("` and `")"` respectively.

continued on next page

continued from previous page

Macro category Macro name(s)	*Node/Section in [Stal 92]* Comment
	Uninitialized Data
ASM_OUTPUT_COMMON (STREAM, NAME, SIZE, ROUNDED)	Outputs to STREAM an assembler directive to reserve an uninitialized memory area of SIZE bytes with the global symbol NAME.
ASM_OUTPUT_LOCAL (STREAM, NAME, SIZE, ROUNDED)	As ASM_OUTPUT_COMMON (STREAM, NAME, SIZE, ROUNDED) but for a local symbol, local to that file.
	Label Output
ASM_OUTPUT_LABEL (STREAM, NAME)	Outputs to STREAM an assembler definition for a symbol with NAME. Unless the assembler has more than one type of label, this macro is used.
ASM_GLOBALIZE_LABEL (STREAM, NAME)	Outputs assembler directives to make the symbol NAME global.
ASM_OUTPUT_LABELREF (STREAM, NAME)	Outputs a reference of NAME to STREAM for the assembler.
ASM_OUTPUT_INTERNAL-_LABEL (STREAM, PREFIX, NUM)	Outputs a compiler-internal symbol generated from PREFIX and NUM; preferably it should be generated using a convention that excludes it from the symbol table in the output from the assembler.
ASM_GENERATE-_INTERNAL_LABEL (STRING, PREFIX, NUM)	As ASM_OUTPUT_INTERNAL_LABEL, but to be stored in STRING, not a stream.
ASM_FORMAT_PRIVATE-_NAME (OUTVAR, NAME, NUMBER)	Generates a privatized version of NAME with sequence NUMBER. Used for example for symbols for `static` variables in functions.
	Instruction Output
REGISTER_NAMES	A C `char *`-vector containing the names for the registers.
PRINT_OPERAND (STREAM, X, CODE)	Outputs the assembler code part equivalent to the internal representation for an instruction operand. CODE is a modifier that describes the output format and is specific for the port.

continued on next page

continued from previous page

Macro category Macro name(s)	*Node/Section in [Stal 92]* Comment
PRINT_OPERAND-_ADDRESS (STREAM, X)	As PRINT_OPERAND, but for memory references; since they might need special formatting.
	Dispatch Tables
ASM_OUTPUT_ADDR_DIFF-_ELT (STREAM, VALUE, REL), ASM_OUTPUT_ADDR_VEC-_ELT (STREAM, VALUE)	Either of these two macros is defined. An entry in a jump-table to be used in `switch` { `case ...:` }-tables should be output to STREAM. Define the ..._DIFF_ELT variant if the table should consist of differences between the table start and the target label, and the ..._VEC_ELT variant otherwise.
	Alignment Output
ASM_OUTPUT_SKIP (STREAM, NBYTES)	Outputs assembler directives to fill NBYTES byte with zero in code or initialized data.
ASM_OUTPUT_ALIGN (STREAM, POWER)	Outputs assembler directives to align the next data to be output at a multiple of 2^{POWER}.
	DBX Options
	GCC knows of major debugging formats such as DBX, SDB, DWARF and XCOFF. No extra definitions than the debugging format is normally needed.
DBX_DEBUGGING_INFO	Defined if the DBX debugging format should be used.
Fundamental machine properties	*Storage Layout*
{BITS, BYTES, WORDS, FLOAT}_BIG_ENDIAN	These macros evaluate to non-zero if the target is of big-endian-type for the particular data sizes.
(MAX_)BITS_PER_{UNIT, WORD}, POINTER_SIZE	States the sizes of machine-accessible data expressed in bits.
FUNCTION_BOUNDARY	This is defined to whatever alignment the start-address of a function must have, in bits.

continued on next page

continued from previous page

Macro category Macro name(s)	*Node/Section in [Stal 92]* Comment
BIGGEST_ALIGNMENT	The biggest alignment required for any data type (not counting FUNCTION_BOUNDARY).
STRICT_ALIGMENT	A non-zero value states that the machine cannot access data on other alignment than the preferred, as stated either implicitly by the size of the type, or explicitly via the ... _ALIGMENT macros.
{BIGGEST, BIGGEST_FIELD, CONSTANT, DATA}- _ALIGNMENT, {PARM, STACK, EMPTY_FIELD, STRUCTURE_SIZE}- _BOUNDARY	Defined if necessary, or for optimization purposes.
MAX_FIXED_MODE_SIZE	If the machine does not have instructions that can move data larger than what fits in one register, i.e. it does *not* have a double-(or-more)-register move, I would recommend to define this macro to that of the register size in bits. The default is the size of DImode; 64 bits for a 32-bit machine.
TARGET_FLOAT_FORMAT	If not defined, IEEE_FLOAT_FORMAT is assumed, which should be used unless the machine does not have another, native format.
	Register Basics
FIRST_PSEUDO_REGISTER	The number of registers in the architecture (the next number after the last index). Note that this does not have to include special registers, unless they can express something useful to GCC.
FIXED_REGISTERS	Registers that are not available for general use.
CALL_USED_REGISTERS	Registers that must be assumed changed after a function call. Must include FIXED_REGISTERS.

continued on next page

continued from previous page

Macro category Macro name(s)	*Node/Section in [Stal 92]* Comment
	Allocation Order
REG_ALLOC_ORDER	An array of numbers, representing the preferred order of allocation of the registers. The default value is the sequence {`0, 1, ... FIRST_PSEUDO_REGISTER-1`}
	Values in Registers
HARD_REGNO_NREGS (REGNO, MODE)	Number of registers needed to hold a value of MODE, counted from register number REGNO.
HARD_REGNO_MODE_OK (REGNO, MODE)	Non-zero if a value of MODE fits into one or more registers starting with REGNO.
MODES_TIEABLE_P (MODE1, MODE2)	States whether it is desirable to choose the same register to avoid move-type instructions between different modes. This is desirable for compact code.
	Register Classes
NO_REGS, ..., GENERAL_REGS, ALL_REGS	Registers are classified in one or more *register classes*. One register can belong to more than one class. There must be at least three named classes, NO_REGS, GENERAL_REGS and ALL_REGS. These last two can be the same. This is all that is needed for a machine with one single type of registers. The register classes must be expressed by the type `enum reg_class`.
N_REG_CLASSES	The number of register classes.
REG_CLASS_NAMES	The names of the register classes, an array of `char *`:s.
REG_CLASS_CONTENTS	An array representing the register set contents of each register class. For the trivial one-type-register case, this is just {`0`, $2^{FIRST\_PSEUDO\_REGISTER} - 1$}.
REGNO_REG_CLASS (REGNO)	The smallest register class containing register REGNO.
BASE_REG_CLASS	A register class capable of holding an address.

continued on next page

continued from previous page

Macro category Macro name(s)	*Node/Section in [Stal 92]* Comment
INDEX_REG_CLASS	A register class capable of holding an index. The index is used in an addressing-mode, adding it to a base address, possibly while it is multiplied by a size-factor. The base address may be kept in a BASE_REG_CLASS register or it may be a constant.
REGNO_OK_FOR_BASE_P (NUM)	Non-zero if register NUM is valid for use as a base register.
REGNO_OK_FOR_INDEX_P (NUM)	Non-zero if register NUM is valid for use as an index register.
SMALL_REGISTER-_CLASSES	Must be defined if the architecture has very few members of a major register class, like three or less for BASE_REG_CLASS. It is not recommended to define this macro unless the compilation of some large test program (or your final intended application) fails with a fatal "fixed or forbidden registers spilled" error message. As an example, the i386 port has this macro defined.
CLASS_MAX_NREGS (CLASS, MODE)	Maximum value of HARD_REGNO_NREGS (CLASS, MODE) for all registers in CLASS.
	Addressing Modes
HAVE_{PRE, POST}_{INC, DEC}REMENT	Defined if the machine has that type of side-effect addressing-mode.
CONSTANT_ADDRESS_P (X)	A non-zero value if X is a valid constant address in the internal format. If it is not, then the address cannot be directly used in the code, and has to be reached in other ways, like indirect through a table associated with the function.
MAX_REGS_PER_ADDRESS	The maximum number of registers that the architecture can use in a memory address.
GO_IF_LEGITIMATE-_ADDRESS (MODE, X, LABEL)	The most important macro of the addressing-mode recognition. If the address kept in X is valid for MODE, then the macro shall `goto LABEL`.

continued on next page

continued from previous page

Macro category Macro name(s)	*Node/Section in [Stal 92]* Comment
REG_OK_FOR_BASE_P (X)	Like REGNO_OK_FOR_BASE_P (NUM) but has a slightly different use, and X is in internal format.[5]
REG_OK_FOR_INDEX_P (X)	Analogous to REG_OK_FOR_BASE_P.
GO_IF_MODE_DEPENDENT_ADDRESS (ADDR, LABEL)	If ADDR behaves differently depending on the mode, the macro shall `goto LABEL`. This often happens for e.g. post-increment addressing-modes.
	Condition Code
NOTICE_UPDATE_CC (EXP, INSN)	If the architecture uses a special condition-code register which is updated as a part of the execution of arithmetic instructions, then this macro should define that effect.
	Costs
CONST_COSTS (X, CODE, OUTER_CODE)	Part of a `switch() {...}`-statement, containing `case`s for CONST_INT, CONST, SYMBOL_REF, LABEL_REF and CONST_DOUBLE. The code should test for constants, and contain `return`-statements specifying the costs of different constants, relative to the macro COSTS_N_INSNS (N), which gives the value 2 for $N = 1$.
RTX_COSTS (X, CODE, OUTER_CODE)	This macro has a decent default value, but to make GCC generate an optimal code sequence, you may need to define this macro.
ADDRESS_COST (ADDRESS)	Like RTX_COSTS, but for the cost of the addressing-mode in ADDRESS.
REGISTER_MOVE_COST (FROM, TO)	Specifies the cost, relative to COSTS_N_INSNS (N), for a register move from class FROM to class TO. The default is 2.

continued on next page

[5]Yes, some of the macros, like this one, certainly seem redundant with respect to combinations of other macros, and this is all a bit confusing at first.

continued from previous page

Macro category Macro name(s)	*Node/Section in [Stal 92]* Comment
MEMORY_MOVE_COST (M)	Like REGISTER_MOVE_COST, but for the cost of moving something of mode M between a register and memory. The default cost is 2.
BRANCH_COST	The cost of a branch instruction. The default is 1. A higher cost makes GCC generate alternative an instruction sequence when possible.
ABI	*Storage Layout*
PARM_BOUNDARY	The memory alignment in bits of parameters passed on stack.
	Type Layout
{INT, SHORT, LONG, CHAR, FLOAT, DOUBLE, LONG_DOUBLE, WCHAR}_TYPE_SIZE	The size in bits of that *C* type.
	Frame Layout
STACK_GROWS-_DOWNWARD	Defined if the stack-pointer goes to a lower address for a new stack-frame. This is true for most ports.
FRAME_GROWS-_DOWNWARD	Defined if local variables are at lower addresses than where the frame-pointer points. This is true for most ports.
ARGS_GROW_DOWNWARD	Defined if the function arguments from right to left in a function call are found at decreasing addresses, when passed on stack. This is not true for most ports.
STARTING_FRAME_OFFSET	The offset from the frame-pointer to the first local variable on stack, if there is one. For compact code, try to make this constant, and if possible, zero.
FIRST_PARM_OFFSET (FUNDECL)	Offset from the frame-pointer to the first argument passed to the function. This value may depend on FUNDECL.

continued on next page

continued from previous page

Macro category Macro name(s)	*Node/Section in [Stal 92]* Comment
STACK_DYNAMIC_OFFSET (FUNDECL)	Offset from the stack-pointer to a dynamically allocated object (like variable-sized arrays and objects allocated with the GCC-builtin `alloca()`-function. This is not used for "normal" *C*-code.
	Frame registers
STACK_POINTER_REGNUM, FRAME_POINTER-_REGNUM	The corresponding register numbers. The stack-pointer must be one of the FIXED_REGISTERS.
	Elimination
FRAME_POINTER-_REQUIRED	Non-zero if the current function *must* have a frame-pointer, which is not desired. Note that GCC may find that the function needs a frame-pointer regardless of this value.
INITIAL_FRAME_POINTER-_OFFSET (DEPTH-VAR)	Used after the function prologue[6] to store the difference between the frame-pointer and the stack pointer into DEPTH-VAR. If FRAME_POINTER_REQUIRED is always non-zero for all functions, then the stored value is unimportant.
	Stack Arguments
PROMOTE_PROTOTYPES	Defined if function arguments should be passed as if the function did not have a prototype, i.e. arguments of "float"-type are passed as `double`, and arguments of an integral type smaller than `int` are passed as `int`s.

continued on next page

[6]See page 36.

continued from previous page

Macro category Macro name(s)	*Node/Section in [Stal 92]* Comment
PUSH_ROUNDING (NPUSHED), ACCUMULATE_OUTGOING-_ARGS	There are three strategies for allocating stack space for function arguments in function-calling functions. When there is a *push*-type instruction, you can choose to push the arguments just before the function call, and de-allocate them after the call. Or, as a second alternative, the area may be allocated before the call, and the arguments stored into that area be de-allocated after the call. The third alternative is to allocate the area at the beginning of the function, and de-allocate it at the end of the function. This last alternative is recommended if there is no fast push-instruction, since it makes for the least overhead in code and execution. If the first alternative should be used, define PUSH_ROUNDING to return the actual absolute difference of the stack-pointer after pushing NPUSHED bytes. Do not do this if the second alternative should be used. Likewise for the third alternative, in which case ACCUMULATE_OUTGOING_ARGS should be defined instead. This causes the variable `current_function_outgoing_args_size` to contain the number of bytes to allocate, used in the function prologue.[7]
RETURN_POPS_ARGS (FUNDECL, FUNTYPE, STACK-SIZE)	The called function may itself de-allocate the allocated stack-space for its arguments. Define this macro to return the number of bytes that the function described by FUNDECL and FUNTYPE de-allocates. May be the constant zero.

continued on next page

[7]See page 36.

continued from previous page

Macro category / Macro name(s)	*Node/Section in [Stal 92]* / Comment
	Register Arguments
	If each function parameter is passed either entirely on stack or entirely in registers, then only the following macros are necessary. In other cases, see [Stal 92].
CUMULATIVE_ARGS	A type used for accumulating information about arguments to a function, for function arguments going from left to right. You will need this for e.g. keeping count of the register numbers for parameters passed in registers. It is not necessarily used, if all arguments are passed on stack.
INIT_CUMULATIVE_ARGS (CUM, FNTYPE, LIBNAME, INDIRECT[8])	Initializes the variable CUM which is of type CUMULATIVE_ARGS. After use of this macro, CUM should be ready to be used for analyzing the first function argument.
FUNCTION_ARG (CUM, MODE, TYPE, NAMED)	Returns an expression in internal representation of a register containing the "next" argument as described by MODE, TYPE and NAMED. Previous arguments are accumulated in CUM. If the argument is *not* passed in a register, the value must be zero, cast to the internal type.
FUNCTION_ARG_PASS_BY-_REFERENCE (CUM, MODE, TYPE, NAMED)	If the function argument must be passed by reference, because e.g. it is too large for a register, this macro must return non-zero.
FUNCTION_ARG_ADVANCE (CUM, MODE, TYPE, NAMED)	Updates the information in CUM for use with the next function argument.
FUNCTION_ARG_REGNO-_P (REGNO)	Must be non-zero if REGNO is one of the argument-passing registers, if any.
	Scalar Return
FUNCTION_VALUE (VALTYPE, FUNC)	Returns a description in internal format of where function return values are found after the call. Not used for aggregate return types.

continued on next page

[8]This parameter was added after version 2.7.2 and is the only necessary change to the CRIS target description for version 2.8.1

continued from previous page

Macro category Macro name(s)	*Node/Section in [Stal 92]* Comment
LIBCALL_VALUE (MODE)	Same as FUNCTION_VALUE, but for the return values of library calls returning values of MODE. The call is for e.g. an arithmetic function and is not necessarily visible in the source code.
FUNCTION_VALUE_REGNO-_P (REGNO)	Analogous to FUNCTION_ARG_REGNO_P, this macro returns non-zero if REGNO may be the number of a register used to return values from functions. Only the first of several registers returning parts of the same value need to be recognized.
	Aggregate Return
STRUCT_VALUE_REGNUM, STRUCT_VALUE	Either of these macros must be defined, used for functions returning structures. If the location of where to put the returned structure is passed to the called function in a register, STRUCT_VALUE_REGNUM specifies the number of that register. Otherwise, the location of where to return the structure is specified in internal representation by STRUCT_VALUE.
	Function Entry
FUNCTION_PROLOGUE (FILE, SIZE)	This macro outputs to FILE the assembler code for the beginning of a function, such as saving registers and setting up the frame-pointer. Specific variables (see [Stal 92]) are provided as indicators of the function type, number of arguments, modified registers etc.
FUNCTION_EPILOGUE (FILE, SIZE)	This macro outputs to FILE the assembler code for the end of a function. If the code profits from having multiple return locations, this macro should recognize those situations and no code should be output. Instead, the standard pattern `return` should output the necessary instructions.

continued on next page

continued from previous page

Macro category Macro name(s)	*Node/Section in [Stal 92]* Comment
	Varargs To implement functions with a variable number of arguments is easy, if the ABI specifies that functions are called with all parameters passed on stack. That means that parameters are always handled equally and can be viewed as an array. This simplifies the implementation and reduces code overhead for this kind of functions. However, if some parameters are passed in registers, it's a bit different. If the parameters are passed in different registers depending on argument type and position, then it's definitely a hassle. Of course, functions with a variable number of arguments are rare enough that there is no specific need to optimize for them.
EXPAND_BUILTIN-_SAVEREGS (ARGS)	If defined, this macro is used to generate code when `__builtin_saveregs()` is used.[9] The code will be expanded just *after* the function prologue. It will always be expanded just after the function prologue, no matter where in the function the `__builtin_saveregs()` call is placed.
SETUP_INCOMING-_VARARGS (ARGS_SO_FAR, MODE, TYPE, PRETEND_ARGS_SIZE, SECOND_TIME)	An alternative to using `__builtin_saveregs()`. With this macro, information can be stored in a target-specific variable, which can then be used in the function prologue to save the parameters passed in registers so they can be easily accessed. Note that [Stal 92] states a different use; to modify the passed arguments so they look like stack-passed parameters. However, it can only be used that way, if the parameter-modification is possible `after` the function prologue.

continued on next page

[9] See page 39.

continued from previous page

Macro category Macro name(s)	*Node/Section in [Stal 92]* Comment
	Misc
CASE_VECTOR_MODE	The size of each element in a `switch { case ...: }`-table.
CASE_VECTOR_PC-_RELATIVE	Defined together with ASM_OUTPUT_ADDR_DIFF_ELT[10] if the elements in the `switch { case ...: }`-table hold offsets from the table beginning to the "case"-code.
Machine description support	*Register Classes*
REG_CLASS_FROM_LETTER (CHAR)	The register class corresponding to the register letter CHAR .[11]
CONST_OK_FOR_LETTER_P (VALUE, C)	Non-zero if VALUE fits in the constant-values set represented by the letter C.[12]
CONST_DOUBLE_OK_FOR-_LETTER_P (VALUE, C)	Non-zero if VALUE fits in the set of floating-point-values represented by the letter C.
EXTRA_CONSTRAINT (VALUE, C)	Non-zero if C represents one of the special operands (other than floating-point, integer constant or floating-point), represented by the letter C.

There are a few standard declarations that must be present, besides the macros and their support:

- `extern int target_flags;`
- `enum regclass {NO_REGS, ... ALL_REGS};`[13]

3.2.2 Variable arguments

The implementation of variable arguments is compiler- and target-dependent. It is most often implemented as macros, using compiler extensions, in the files corresponding to `<varargs.h>` and `<stdarg.h>`. For GCC, `va_start()`, `va_dcl()`, `va_arg()` and `va_end()` should be defined using the following built-in functions:

[10]See page 27.
[11]See page 29 and 42.
[12]See page 43.
[13]See page 29.

__builtin_saveregs() If the variable-argument function does not itself save registers in the function prologue,[14] this built-in function is used as a marker to tell GCC to insert the contents of the macro EXPAND_BUILTIN_SAVEREGS.[15] That is, the definition is something like:
`#define va_start(va, lastarg) __builtin_saveregs(), va =` __builtin_next_arg().

__builtin_args_info (CATEGORY) Used if function parameters end up in different types of registers. This built-in function should be used in the definition of `va_start()`, to retrieve information of which registers have been used for the named parameters.

__builtin_next_arg (LASTARG) Returns the address of the first unnamed argument. To be used in the `va_start` definition.

__builtin_classify_type (OBJECT) To be used together with `sizeof()` and the builtin function `__alignof__()` to find out where the next argument may be located.

3.2.3 The *C* file: `tm.c`

Many of the macros in the `tm.h`-file will be too complex and hard to debug if their definitions are just raw code. Instead, it is customary to have a function with the same name as the macro, but in lower-case letters only, so *that* particular function can be debugged, instead of a macro expansion in several files. For example, the macro that describes the condition-code setting effects of various functions is named `NOTICE_UPDATE_CC(EXP, INSN)`. So, if this is implemented as a function, the definition would be:

```
#define NOTICE_UPDATE_CC(EXP, INSN) notice_update_cc(EXP, INSN)
```

and of course next to the macro definition, there would be a declaration:

```
extern void notice_update_cc();
```

That traditional K&R function-declarations are used in this file to get maximum portability.[16]

The `md` machine description file often needs support-functions too, for example for straightforward descriptions of what operands an instruction can take.[17]

3.2.4 The machine description: `md`

The machine description is written in a machine description format called *RTL*, which is closely related to the internal data representation, *RTX*. An instruction description consists of *instruction template patterns* usable for both instruction *generation* and instruction *matching*. At compilation, first a basic sequence of

[14] See page 36.
[15] See page 37.
[16] Or better, in the focus of recent improvements: Full ANSI declarations enclosed in macros that optionally make them visible if the host compiler can handle argument declarations.
[17] See page 41.

instructions is generated, and later passes combines and splits these instructions, trying to match a faster sequence.[18] The `md` file contains instruction definitions, attribute definitions, instruction-splitting descriptions and peephole optimization definitions.

Instruction definition patterns

An instruction has the following generic pattern definition structure:

(`define_`*pattern-type* `"`*(optional) pattern name*`"`
  `[(set (`*target*`)`
    `(`*the operand*`))`
   optionally more setting-effects`]`
  `"`*optional pattern-condition*`"`
  `"`*output template*`"`
  (optionally: `[`*attribute settings*`]` *)*)

For simple examples, see page 44. The parts have the following semantics:

pattern name The pattern does not *have* to have a name; an empty string makes it an anonymous pattern. Other than that, names have to be unique. Some names are reserved for common and mandatory instruction patterns with a predefined behavior and usage. These are called *standard names*.[19] For example, the standard name for addition of register-size operands (SImode) is `addsi3`, for add-single-integer using three operands. The three operands are the two source operands and the result[20] Certain named patterns are mandatory, like the *move*-patterns for register-size operands, subroutine-call instructions, branches and indirect-jump. For a standard operation that has no named pattern, a library function is called.

pattern-type There are two types of pattern descriptions:

define_insn The most common pattern type. This is valid both as a generator and matching pattern.

define_expand Some standard-named patterns are preferably translated into a couple of other instructions rather than a library call. This is called *pattern expansion*. The expansion definition patterns are only used when the operation related to the standard name is called for. They are not recognized when synthesizing complex instructions from simpler ones.

The pattern *name* from a `define_insn` or `define_expand` is turned into a generated function called `gen_`*name*`()`. The functions generated from standard-named patterns are called explicitly from within

[18] See page 53 for the different passes where this happens.

[19] This set increases slightly with each GCC version, as new architectures are added, with specialized instructions and opportunities for optimization when using them.

[20] See [Stal 92] for a complete list of the standard names.

GCC, when needed and defined. GCC knows by itself how to synthesize missing instructions for simple logical and arithmetic operations, using existing instruction patterns of larger or smaller sizes. For example, two word-sized `and` instructions can be used on a dword-sized operand, if the machine has no dword-sized `and` instruction.

the operand A composition of operations of any complexity. For an addition, this looks like `(plus:`*M* `(operand1) (operand2))`, where *M* denotes the machine mode, and the operands of the operation can in turn be composed of other operations. Standard-named patterns have a fix appearance and placement of operands. Therefore `define_expand` does not specify the original pattern, just the resulting patterns.

At the leaves of the resulting operation "tree", there is usually some kind of operand-matching expression. The generic form is `(match_`*operand-type*`:`*M operand-number* `"`*predicate*`" "`*constraints*`")`. The operands and operators of the pattern are normally numbered by increasing numbers from zero, with the first destination operand as zero. The matching expressions can be of four different types:

match_operand The main case: `(match_operand:`*M operand-number* `"`*predicate*`" "`*constraints*`")`. The *predicate* defines what general kinds of operand is allowed. The *constraints* further specify what combinations of operands are allowed.

match_operator To match a set of operators, like `plus`, `and`, `xor`, you can specify a predicate for that set, just as for different operand types. You may be tempted to match an operator-expression with an intricate predicate and set of constraints in a single `(match_operand ...)`-expression, but this will be sub-optimal[21] for the register allocation pass,[22] that has to be decide what registers are allocated for what operand.

The operands should be defined as an array: `(match_operator:`*M operator-number* `"`*predicate*`" [`*operands*`])`, where *operands* are one or more of *the operand* defined in the above (recursively).

match_dup Just the same as the *operand-number*:th operand. For this case, just the *operand-number* part is present; no mode, predicate or constraint. For example: `(match_dup 1)`.

match_op_dup The same as the *operand-number*:th operator, as with match_dup.

The *predicate* is the name of a *C* function returning an integer, zero or one, for a match of the operand (or operator) and mode in its fixed-type arguments. There are several pre-defined predicate functions:

register_operand An operand that is a register.

[21] See page 91.
[22] See page 53.

address_operand An operand that is an address, as matched by `GO_IF_LEGITIMATE_ADDRESS()`.[23]

immediate_operand An operand that is a constant (maybe a constant address).

const_int_operand As `immediate_operand`, but for integer values only.

const_double_operand As `immediate_operand`, but for floating-point values only.

nonimmediate_operand An operand that does *not* match `immediate_operand`.

memory_operand An operand that is in memory.

general_operand Just any valid operand; register, constant or memory.

indirect_operand A `memory_operand` whose address is a `general_operand`.

comparison_operator An operator that is a comparison (equal, greater, less-than etc.)

The *constraints* are a possibly empty string of letters and symbols, one for each operand. They comprise the second level of operand-match description,[24] of which the predicate is the first level. Operators do not have constraints, but their operands often have. Constraints describe the possible mixtures of operands and their relative cost and desirability. The alternatives are separated by a comma. For architectures with different classes of registers, it is common to define a letter for each register class.[25] Certain symbols and letters have predefined meanings. Some of the special symbols relate to that alternative only, others mark a property for the entire operand. The following list covers most of the simpler constraints:

0 . . . 9 The operand for this alternative has to be the same as the operand with the specified number.

r This alternative is a register that is in `GENERAL_REGS`.[26]

g Any operand that fits the `general_operand` predicate matches this alternative.

m Any `memory_operand` matches this alternative.

= The operand is assigned to, and the original value is lost. This is in effect for all alternatives of this operand.

+ This operand is only partly modified and is not completely determined by the assignment of the operand. Marks this effect for all alternatives.

[23] See page 30.
[24] See page 53.
[25] See page 38.
[26] See page 29.

& This symbol marks that this constraint-alternative is partly written before the other input operands are finished reading in, so they can not be the same as this operand.

% This operand can be exchanged for the operand with the next number, for all alternatives. This normally happens for commutative operations, such as `plus` and logical `and`.

I ... P As an example of constraints defined in the target description,[27] these letters are reserved for ranges of constants, defined through the macro CONST_OK_FOR_LETTER_P. They are mostly used for the cases where some constants fit in better or faster code, like the quick-immediate mode in CRIS.

target This is the output of the instruction. If the only effect of the instruction is to set condition codes, then the target is expressed as `cc0`, but normally the condition-code setting is not expressed in the pattern. The target has the same appearance as an operand, but can not be an operation.

more setting-expressions Optionally more `(set ...)`-expressions. Note that they are considered to be executed in parallel, so the output of one can not be used as the input of another.

pattern-condition An optional (default true) condition for when the pattern applies. For a standard name, you must refrain from testing the operands. Only `TARGET_FLAGS` may be tested, except for the pattern `return`. This means that if you have set of similar architectures that has different types of operands depending on which CPU the code is compiled for, you must have a `(define_expand ...)`-pattern for the superset of allowed operand types. Different anonymous patterns can then test the operands and `TARGET_FLAGS` for a valid combination.

The cause for this rule is that a standard-named pattern must be known valid-or-not throughout the compilation. So, only a condition that is constant for the entire compilation is allowed.

output template The specification of assembler output for `define_insn`, or a piece of *C* code to execute to e.g. tailor the operands for `define_expand`. For `define_insn`, the assembler output can be generated in a number of ways:

- If the output template is a piece of *C*-code then it can return a string and call `output_asm_insn(const char *, const rtx [])`. The first character after the double-quote must then be an asterisk: `*`.
- Just the assembler instruction, as a string. No format specification is needed.

[27] See page 38.

- A list of assembler instruction strings, divided by a newline with surrounding whitespace. The first character after the double-quote then has to be an at-sign: @. The number of the matching constraint-alternative decides which list-element is used.

For the assembler instruction-strings, a percent-sign: % followed by a number denotes the place of that operand number. The operand is output by PRINT_OPERAND() or PRINT_OPERAND_ADDRESS(), whichever applies.

attribute settings If any attributes for this instruction are not covered by the defaults,[28] they can be specified in the instruction, using conditions or just a list of the same number of elements and left-to-right order as the constraints.

Some examples of real instruction definition patterns from the CRIS port:

```
(define_insn "xorsi3"
  [(set (match_operand:SI 0 "register_operand" "=r")
        (xor:SI (match_operand:SI 1 "register_operand" "%0")
                (match_operand:SI 2 "register_operand" "r")))]
  ""
  "xor %2,%0"
  [(set_attr "slottable" "yes")])
```

This is a very simple example. It defines a standard-named pattern for the xor-instruction. The mode of the operands is the 32 bits of the register (SImode). It only works for registers, so all operands use the predicate register_operand. Furthermore, the destination operand has to be the same as one of the operands, so the first source operand has the constraint %0 marking that if needed for making a match, it may be exchanged with the next operand (number 2), but it eventually has to be the same as the destination operand, number 0. The instruction has only one single appearance, so the constraints as well as the assembler output is one single item.

The two operands can be swapped because exclusive-or is a commutative operation. The empty pattern-condition shows that the pattern is always valid. The assembler output is simple; just the text xor followed by the two operands, where the destination register is the last one. Operand number 1 is not used, since it must be the same as operand number 0; the destination. The instruction has an attribute that is not covered by the default; the attribute slottable[28] will be yes, and so has to be specified here.

[28] See page 49.

```
(define_expand "negsf2"
  [(set (match_dup 2)
        (const_int -2147483648)) ; 0x80000000
   (parallel [(set (match_operand:SF 0 "register_operand" "=r")
                   (neg:SF (match_operand:SF 1
                                  "register_operand" "0")))
              (use (match_dup 2))])]
  ""
  "operands[2] = gen_reg_rtx(SImode);")
```

This pattern tells GCC how to generate a single-precision (SFmode) negation from existing instructions. Since for IEEE-format this is always done by "flipping" the sign-bit (there is a negative zero), it can be done using a integer exclusive-or-instruction with the bit-pattern 0x80000000 to the operand holding the floating-point number.

The mandatory operands for an `negsf2`-pattern are operand 0 as the destination, and operand 1 as the source operand. These operands are mentioned in the resulting pattern; they are made sure to be in registers by specifying `register_operand` as the predicate. The constraints are not really used, they are there only for reasons of documentation.

The resulting expansion will be two instructions; the first one just sets a register to the value 0x80000000, although the number has to be expressed to the machine-description-reader of GCC in signed decimal notation: -2147483648.[29] The second instruction looks the same as the "original" `negsf2`-instruction, but with a note that tells that it uses the register with the sign-bit-value, despite that it is not mentioned in the normal operands. The `use`-operation is used to mark such needs, when it cannot be logically deduced through the pattern. The *C* code "modifies" the pattern by generating the pseudo-register[30] for the first instruction.

There are two reasons for not just matching `negsf2` with a pattern that pretends to be a real `negsf2`-instruction but actually outputs the two instructions above. First, there may actually be a register with the value 0x80000000 handy, a case which GCC would optimize. Second, the exclusive-or instruction will fit in a delay-slot, so the two instructions are better off each on their own. Still, why not expand the second instruction into an `xor`-pattern? Well, that would confuse GCC, since the mode of the data would then no longer be SFmode, but instead suddenly SImode for no apparent reason.

[29] This number is a portability abomination, which just seems to work on several platforms at the moment. A more portable solution is sought after.

[30] See page 54.

```
(define_insn ""
  [(set (match_operand:SF 0 "register_operand" "=r")
        (neg:SF (match_operand:SF 1 "register_operand" "0")))
   (use (match_operand:SI 2 "register_operand" "r"))]
  ""
  "xor %2,%0"
  [(set_attr "slottable" "yes")])
```

This is an anonymous pattern that matches what was generated by the `define_expand` above. The value 0x80000000 is not present here other than as implied via the `use`-operation. GCC will not accidentally generate a negation through *this* pattern instead of the one in the `define_expand`, since the standard-named patterns are used as generators, and unnamed patterns are only used for recognition and combination of simpler patterns. Above all, the `use` construct does not exist other than together with other patterns, and are not generated other than as explicitly stated above.

But these examples are not really *that* profitable in real life — there are not enough floating-point negations in the typical intended target application to excuse for the extra complexity in the machine description.

```
(define_insn ""
  [(set (cc0)
        (compare
         (match_operand:SI 0 "register_operand" "r,r")
         (match_operator:SI 2 "extend_operator"
           [(match_operand:HI 1 "memory_operand" "Q>,m")])))]
  ""
  "cmp%e2.w %1,%0"
  [(set_attr "slottable" "yes,no")])
```

This anonymous pattern is a little bit more intricate. It matches comparisons to a register for the whole size, SImode. The predicate `extend_operator` matches a zero- or sign-extended word-sized (HImode) operand that must be located in memory — a register is not allowed for this operand.

The assembler output uses the output-operand modifier-letter `e` to operand 2, the operator. This modifier-letter makes `PRINT_OPERAND()` output the letter `s` for sign-extend, or `u` for zero-extend, depending on what operator is in operand 2.

There are two "constraint" alternatives, specified in order of increasing cost from left to right. The first alternative matches when the operand in memory is of a simple kind of addressing-mode, for which the attribute `slottable` should be `yes`. The second constraint matches any memory operand, but since the alternatives are tried in order left to right, only the ones that did *not* match the first will end up here. These left-over operands will get the value `no`.

The usability of this pattern depends on the ability of the instruction combination pass[31] to combine a sign- or zero-extend operation with a comparison.

[31] See page 53.

Instruction splitting

Sometimes the performance of the code can be improved by splitting up complex instructions into smaller ones. This happens for architectures with instructions with delayed or prolonged execution; when another instruction with some restrictions can be executed during the otherwise wasted delay.

To profitably split up a complex instruction, there has to be a pattern to split it up into a set of smaller instructions, and another instruction with a "delay", where a part can fit, for the sake of data dependence and target-dependent constraints. The total set should be shorter or faster than the complex instruction plus the overhead of the unfilled delay-slot. GCC splits instructions speculatively and does not re-unite the result of the split instruction if it cannot be used, so it's best if the result of the split is as fast and short as the original.

The generic format of a "split"-definition description is:

```
(define_split
  [(set (target)
        (the operand))
   optionally more setting-patterns]
  "optional pattern-condition"
  [(replacement-setting-pattern)
   optionally-more-replacement-setting-patterns]
  "optional-preparation-statements")
```

If the condition yields true and after the preparation statements have been executed, then the first bracketed list of setting-patterns is replaced by the second list. Note that as opposed to the `define_expand` and `define_insn` patterns, the list is considered executed in sequence, not in parallel.

Example:

```
(define_split
  [(set (match_operand 0 "register_operand" "=r")
        (match_operand 1 "indirect_operand" "m"))]
  "GET_MODE_SIZE(GET_MODE(operands[0])) <= UNITS_PER_WORD
   && (GET_CODE(XEXP(operands[1],0)) == MEM
       || CONSTANT_P(XEXP(operands[1],0)))"
  [(set (match_dup 2) (match_dup 4))
   (set (match_dup 0) (match_dup 3))]
  "operands[2] = gen_rtx(REG,Pmode,REGNO(operands[0]));
   operands[3] = gen_rtx(MEM,GET_MODE(operands[0]),operands[2]);
   operands[4] = XEXP(operands[1],0);")
```

This pattern recognizes the `move.`*S* `[`*address*`],`*rX* instruction, which can be split up at no extra cost into `move.d` *address*`,`*rX* and `move.`*S* `[`*rX*`],`*rX*.

The predicates check the general appearance of the pattern, and that *address* is a memory operand whose address is a `general_operand`.

Then the pattern-condition further restricts the pattern, making sure that the operands have a mode that fits in a register, and checks *address* to be a constant or another memory reference; something that can be split up profitably.

The preparation statements takes the destination register and uses it as a temporary register for operand 2, then creates the register indirection for operand 3, setting the mode-size of the reference right. The *address* operand goes in operand 4. Note that no new registers are actually generated, to allow this split to occur both before and after register allocation is finished.[32]

As with `define_expand`, the constraints are not used, but are put in for the purpose of documentation.

Peephole optimizations

Some instruction sequences are not easily optimized from analysis of the data flow. Also of course, GCC might miss something. Then a `define_peephole` can be used as a final resort, to describe how to optimize a given sequence of instructions. The generic format is the same as for `define_expand` and `define_insn`, but without a name. For example:

```
(define_peephole
  [(set (match_operand:SI 0 "register_operand" "=r")
        (plus:SI (match_operand:SI 1 "register_operand" "0")
                 (match_operand:SI 2 "const_int_operand" "n")))
   (set (match_operand 3 "register_operand" "=r")
        (mem (match_dup 0)))]
  "GET_MODE_SIZE(GET_MODE(operands[3])) <= UNITS_PER_WORD
   && REGNO(operands[3]) != REGNO(operands[0])
   && (INTVAL(operands[2]) >= -128 && INTVAL(operands[2]) < 128)"
  "move.%s3 [%0=%1%S2],%3")
```

This pattern matches the admittedly rare case, when we have an `add.d` *n*,*rX* next to a `move.`*S* `[`*rX*`],`*rY* instruction. This can be transformed into one instruction, `move.`*S* `[`*rX*`=`*rX*`+`*n*`],`*rY* which is profitable as being a shorter instruction sequence by one word, if $-128 \leq n \leq -64$ or $63 \leq n \leq 127$ (with no difference in speed or size if $-64 \leq n \leq 63$). The above case should be caught by other optimizations, but may happen where the offset or registers has changed during register allocation or frame-pointer elimination.

The condition checks for an allowed size for the operands, checks that the final destination register is not the same as the register used for indirect access, and finally checks the allowed offset range. The destination for the side-effect addressing-mode must not be the same as the destination for the main operation; because both operations are supposed to happen simultaneously, the result would then be undefined.

If a sub-optimal instruction sequence is observed in the resulting assembler code, efforts should normally go in other machine-description areas than

[32] If you *must* allocate new registers for the pattern to work, test the global variable `reload_completed` in the pattern-condition. It will be non-zero when no new registers can be allocated.

peephole optimizations; namely instruction combination using anonymous instruction patterns, instruction splitting, and better discrimination of operands and their cost. The peephole optimizations are only meant for cases where no other possibility of optimization exists, since the effect is limited to instances where the bad code is accidentally produced in sequence, without other instructions in-between. GCC makes no attempt to rearrange instructions to match the peephole optimizations.

Attribute definitions

To simplify the machine description, different *attributes* can be defined to logically group together instructions. For example, it might be usable to identify different instruction types such as arithmetic, jump or move instructions as groups, if all instructions within the group modify the condition-code-register in the same way. To minimize the overhead with defining the attributes, defaults can be defined in various ways.

The generic format is:

`(define_attr` *name list-of-values default*`)`

For CRIS, only the following attribute is present:

```
(define_attr "slottable" "no,yes,branch" (const_string "no"))
```

It defines an attribute `slottable`, used to tag which instructions can fit in a branch-delay-slot. It can have one of the three specified values, and the default is `no`. The `branch` case, basically equivalent to `no`, is meant to identify branch instructions and the `ret` instruction, which have a delay-slot and cannot themselves be put into a delay-slot.

The default-case can be a much more complicated expression than the constant above, depending on other attributes etc. The attributes of instruction patterns that are not handled by the default-case, can be set in the *attribute settings* part of those instruction patterns. The most common expression is then to just set an attribute to a constant value, or to one that depends on what constraint-alternative that matched,[33] but settings can depend on operands or on other attributes of that instruction pattern.

An attribute named `length` is reserved for special semantics to be used when the length of the instruction needs to be approximated.

The attribute of an instruction can be accessed from *C*-code, for use at assembler output or by support-functions.

Delay definitions

Delays at program flow instructions, such as branch instructions, have their own description mechanics in GCC. It will try to fill the delay-slots to minimize code size and execution time.

[33] See examples starting on page 44.

The generic form of the delay-definition is:

(define_delay *attribute-test-expression*
  [*first-delay-filler first-annul-taken first-annul-not-taken*
   more-optional-delay-filler-expressions-when-taken-or-not-taken])

Which means:

attribute-test-expression This is an expression that works on one or more attributes of an instruction, and yields true when it has a delay-slot-combination of this type.

first-delay-filler An expression that matches instructions that may be put in this delay-slot and always be executed, regardless of whether the branch is taken or not.

first-annul-taken An expression that matches instructions that may be put in this delay-slot and will *not* be executed if the branch is taken.

first-annul-not-taken An expression that matches instructions that may be put in this delay-slot and will *only* be executed if the branch is taken.

more-optional-delay-filler-expressions Optionally, for delays with multiple instruction slots to be filled, a number of sequences of *first-delay-filler*, *first-annul-taken*, and *first-annul-not-taken* expressions can be put here to specify what can be filled in each delay-slot.

If there is no instruction that matches a type, put `(nil)` there instead. For CRIS, branch instructions and the `ret` instruction have a one instruction delay-slot; the attribute `slottable` has the value `branch` for them:

```
(define_delay (eq_attr "slottable" "branch")
  [(eq_attr "slottable" "yes") (nil) (nil)])
```

The single delay-slot can be filled with an instruction where the attribute `slottable` has the value `yes`. Instructions that are one word long and do not access the program counter has this value. No “annulling” of delayed instructions exists for CRIS.

3.2.5 The building process of the compiler

Configuration

GCC can be configured to be used on a lot of systems, compiling for that system or used as a cross-compiler for yet other systems. A configuration process must be run before compiling the compiler, and is implemented as a shell-script called `configure`. It, and its helper scripts, checks whether the system where the configuration runs is supported, and whether the target system (if used as a

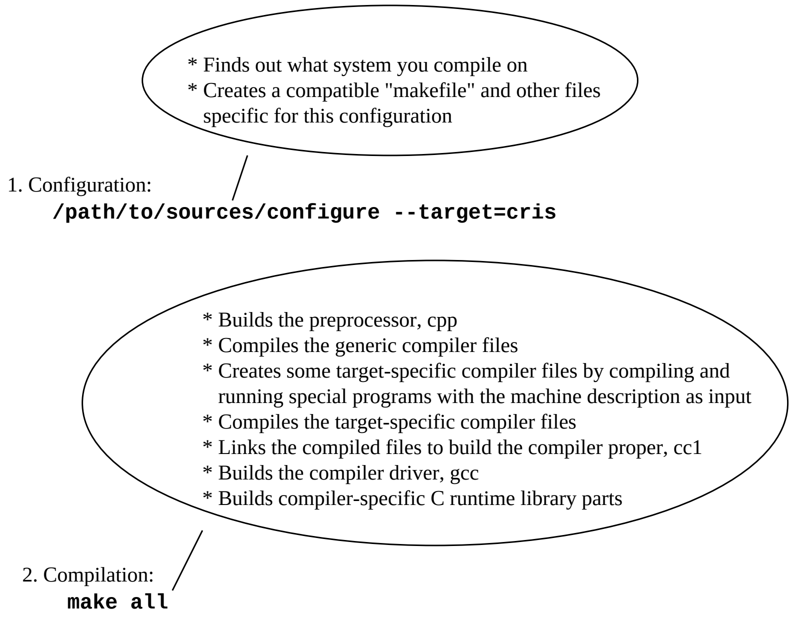

Figure 3.3: The "configure" and "make" compiler-building process

cross-compiler) is supported.[34] There is an easy way to add on a new target or host system; a few lines with another `case` statement is added in a configuration script, with possibly other lines for a nickname.

The systems that are involved, for host and target each, are recognized in a canonical form: *`CPU-company-system`*, which is supposed to completely specify the environment. The *company* part is only meant to be discriminatory when two identically named systems exist. For CRIS, this definition is `cris-axis-none`. The system is specified as `none` as there is no specific system or kernel on which the code must be run.[35]

It is assumed that the installation is performed on some kind of *Unix* system, or that there is some means to run the shell-scripts containing the installation program, and a common denominator of the file format for the `make` utility.[36] System-specific differences in the syntax of the makefile-format are solved by the configuration script by putting the file `Makefile` together from common and system-specific pieces. Files called `tm.h`, `tm.c` and `md` are created, redirecting to the target-specific ones.

[34] Yet another dimension exists; the host-target combination of the compiler can be *configured and compiled* on a *third* type of system.

[35] Complex features such as threads-handling and exceptions support in languages such as *C++* might call for a system to be specified here.

[36] See [make].

Compilation

The program `make` must be present on the system. It supervises the compilation and installation of the compiler.

First, a number of *C* programs, all having names starting with `gen...`,[37] are compiled and executed. They extract information from the machine description, and create one *C*-file each, all given names starting with `insn-`:

insn-attr.h by genattr Definitions for any defined attributes and delay-definitions.

insn-attr.c by genattrtab Functions to access the attributes.

insn-codes.h by gencodes Definitions for named patterns.

insn-config.h by genconfig Some limits specified in the target description, such as the maximum number of constraint alternatives, the presence of a condition-code-register and the maximum number of operands in an instruction pattern.

insn-emit.c by genemit Generator functions for the named patterns (the `gen_`*`name`*`()`-functions), plus some functions for instruction splitting.

insn-extract.c by genextract One big function for taking an instruction and getting the operands ready for operand handling and assembler output.

insn-flags.h by genflags Here is where the conditions for the instruction patterns are used, to specify which standard-named patterns are present.

insn-opinit.c by genopinit Code to initialize tables for various operations as specified by the existence of some standard-named patterns

insn-output.c by genoutput Functions and tables used to output assembler code for the instruction patterns. Tables with constraints and predicates for the instructions.

insn-peep.c by genpeep Functions dealing with peephole optimizations.

insn-recog.c by genrecog Functions to implement a decision tree to determine whether a supplied instruction matches an instruction in the machine description, for general recognition or splitting.

Then the entire code is compiled and linked together to form the major compiler component programs, `gcc`, `cpp` and `cc1`. If the compiler will not be a cross-compiler, the system assembler and linker will be used, and are picked up as specified in the configuration files. If it will be a cross-compiler, the linker, assembler and header files must be installed at the specified location before the compiler installation starts.

[37] Some intermediate programs, `gengenrtl`, and `gencheck` has been added in egcs. While these programs also generate support functions, they do not use the machine description.

Some parts of the C run-time library are created as part of the compilation process. This contains, for example, synthesized functions for arithmetic and logical operations on all sizes of integers that are supported by the compiler but not the actual target architecture, and support for compiler-specific features.

Self-testing

If the system is not a cross-compiler, the just-compiled compiler is then used to compile the compiler (again) to see if there is any difference in the code, both as a test and because it is assumed that GCC produces better code than the system compiler (which may be an older version of GCC) and that it is better to use the GCC that is compiled by itself. The reason is that if it would *not* be better, then there would generally be no gain to install GCC.

Other processing

For a non-cross-compiler, the system header files may need to be adjusted to fit the ANSI and GCC syntax, and to avoid the need to implement specific *C* extensions of the old system compiler. The compilation pass takes care of this automatically. Some standard header files, such as `limits.h` and `float.h` can, depending on the system-specific configuration, be created as part of the building phase. All adjusted and new header files are located in a GCC-specific directory.

3.2.6 Execution of the compiler

The parts that are installed as parts of GCC; the driver `gcc`, the preprocessor `cpp` and the compiler proper, `cc1`, are executed in the order described in chapter 3.1.1.

The preprocessor, cpp

There is not much to say about this program, it is mostly system-independent except for the location of the target-system header files. Only a few defaults are specified in the `tm.h` file. Any system-specific macro definitions are in fact supplied from the compiler driver, using command-line options.

The compiler proper, cc1

All compiler passes are repeated for every function in the source code. Each function is parsed and the internal RTX representation is generated. It is decided whether to have it lying around for function in-lining purposes, or to emit assembler code for it right away.

When a function is emitted as assembler code, it goes through several optimization and code-generating passes. Two of these passes are of greater interest with respect to the machine description: The instruction combination pass and the register allocation pass.[38] If optimization is not desired, the instruction combination pass is skipped, and the register allocation scheme reverts to a simple kind.

At the first stage, there are no allocated "real" registers other than those specifically mentioned in the instruction description patterns. Each time a new temporary result is created (as the result of an operation or as otherwise needed for instructions to match), a new *pseudo-register* is created. The pseudo-registers are treated as hardware registers, but generated with registers number beyond the specified last register number for the hardware registers. This makes it possible to take the incremental approach that all values will fit in registers, and leave it for later to take care of which pseudo-registers will actually be machine registers, and which may end up being located on stack, have to be stored and loaded into real registers[39] for access.

When register allocation starts, all pseudo-registers are analyzed to find out their use. The result of the analysis tells which pseudo-registers have disjoint lifetimes; that is, which can be merged to use the same hardware registers. It also tells which pseudo-registers are used in such a way that they are most profitably mapped to hardware registers.[40] The constraints are especially important in this pass, as a tool to determine the relative cost for the register allocation. Before this pass, the instruction pattern constraints are not used.

Instruction combination uses a previous *data flow* analysis pass to determine which computations can be grouped together or combined in a single instruction. The actual combiner has very little knowledge about the target system. Instead, it relies heavily on the instruction-recognizing decision-tree function `recog()`, to try and see if combinations of instructions that would be useful with this code, really exist on the target architecture.

Instruction splitting occurs at various passes: as needed by the instruction combiner when trying to combine the split instructions with other instructions and by the delayed branch scheduler.[41]

Delayed branch scheduling occurs at a late stage in the compilation. No other instruction processing than instruction splitting is performed at this stage or after. So all patterns, including these patterns resulting from `define_splits` must satisfy their constraints and not generate new temporary registers after register allocation is completed.

The last thing that happens before assembler output is peephole optimization.

Any detected anomaly in the compilation state (which should *not* be caused by errors in the code being compiled), makes the compiler call `abort()`. This is seen in error messages as "internal error ... signal 6". Actually, this simplifies

[38] This is in fact several passes — including the new "address-of" pass added in 2.8.1 and the "reg-move" pass added in egcs 1.0 — that are allocating registers for different classes of values. The distinction visible to the machine description is a change of state where register allocation is not yet started, in progress or completed.

[39] Also known as *register spilling*.

[40] This method is described in [RedDragon, pages 541–546].

[41] For more complex architectures than CRIS, there are other cases too, such as when scheduling instructions for function units.

debugging, if the environment it set up to allow `core` files to be written, as the `core` file reveals which of the large amount of `abort()` calls in the code was the one that was triggered.

Chapter 4

The CRIS port

I could not follow the guidelines in chapter 6.2 myself, since they did not exist, and no guide except [Stal 92] was known.[1]

4.1 Preparations

A first try had already been attempted at a CRIS port for GCC version 1. It had scratched the surface, but was put on ice and later abandoned. I decided to start over from the beginning, after consultations with the project lead.

I first read [Stal 92] a couple of times until I understood the structure of GCC. To get a hunch of the solutions to common problems, I studied some of the ports to architectures I knew something about, like for the MC68K, i386, Vax and NS32K. I then realized the lack of defined structure in the would-be target-system. I had to invent a parameter-passing scheme for CRIS. This type of scheme is known as an *ABI*. This was the first practical step.

4.2 The target ABI

At the time of this work, no previous system existed on the CRIS architecture; even the CRIS architecture itself was not fixed.

The ABI is not normally a design issue when a compiler is ported, since the system probably has been programmed in a well-defined way before, and an ABI is established, with which the port has to stay compatible. But to invent the ABI at this point was probably just as good, since the ABI and the compiler should generally be streamlined together to reach optimal code effectiveness.

I went with general assumptions and some investigations made at the first porting attempt. As I gained more insight, I modified the ABI, based on observations in the assembler code generated by gcc-cris. It wasn't possible to use

[1] No, the WWW did not exist, and whatever other resources on the Internet showed no trace of any beginners-guide to porting GCC. There still are none, as far as I know.

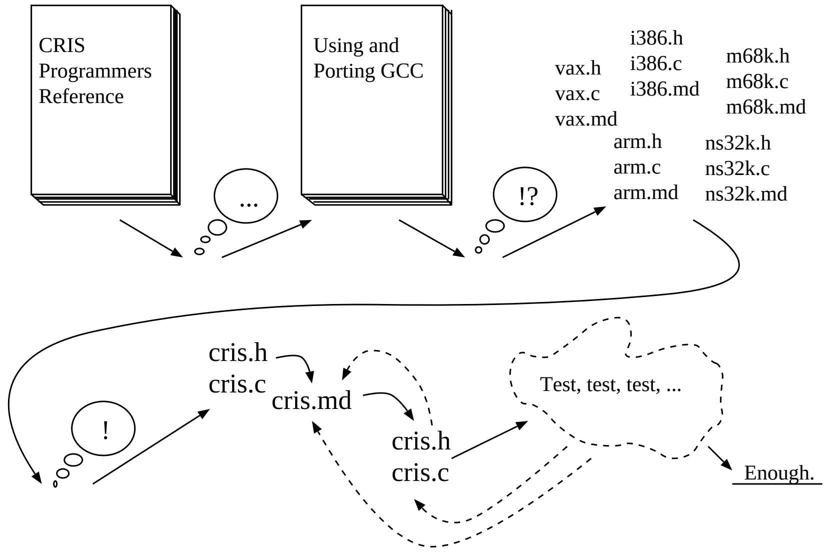

Figure 4.1: The creation of the port (as imagined)

measurements on executable code to optimize the port for a long time, so only brief observations on static code were used to remedy the most blatant mistakes.

4.2.1 Fundamental types

The choice of representation for fundamental types in the system and the ABI is often straightforward. To avoid problems with porting of programs and modules where assumptions are made for certain fundamental types, historical precedence was considered the deciding factor. The following decisions were easy:

- A *char* is represented by a byte, 8 bits. In some commercial compilers, this is an unsigned type. In gcc-cris it is default signed, because this makes it similar to the other integral types, which are all signed by default. Also, it is the historical case for compilers on Unix systems.[2]
- A (*signed*) *short* (*int*) is represented by a word; 16 bits.
- A (*signed*) *int* as well as ...
- ... a (*signed*) *long* (*int*) is represented by a dword, 32 bits.

An *int* could have been represented by 16 bits, but that would not have been the natural representation for a 32-bit machine such as CRIS. Arithmetic and logical instructions for different sizes take the same time when performed on

[2]In *C++* there are three related distinct types; `char`, `unsigned char` and `signed char`.

register-only operands, but the quick-immediate addressing-mode gives the correct condition-code for a 32-bit result, leading to shorter code in the end. No historical precedence exists for actually making the *int* representation shorter than *long*, but there are indications that they should be the same size, as many programs assume you can store and represent pointers and sizes of data with an *unsigned int*.

GCC has a type extension, a type called *long long*. This is represented by a 64-bit entity, consisting of two dwords in dword-little-endian order.

A pointer to any type is represented by an udword holding that address.

An enumeration type, or *enum*, is always a dword. An alternative would have been to make it hold only the smallest entity needed for the definition set. The choice of always-a-dword ensures no surprises when someone changes the possible values of an enum-type in a program, to hold larger values than the previously chosen representation. Of course, this would normally not be noticed in well-written programs.

Floating-point types are not very much used in non-control-oriented embedded systems.[3] There is a standard for floating-point representation: the IEEE-754 standard, which covers representations in 32, 64 and 128 bits, and there is no use in having another, non-standard representation. Thus the first choice was clear for the *float* type: the IEEE-754 32-bit alternative.

For *double* there is a historical assumption that it has a better representation than *float*, and that it can hold an *int* or *long int* value without loss of representation.[4] On the other hand, there is the peculiarity of classic K&R *C*[5] (not in ANSI *C*) that any expression between *float* values is performed by promoting the values to *double*, then evaluating the expression, then possibly truncating the result back to float again when an assignment of the result to a float-type entity is made. Similarly, *float* parameters to functions are promoted to *double* before passing. This historical need for promotion made it easier, at least for performance reasons, to choose the same representation for *float* as for *double*. Also, entities larger than 32 bits can not fit into a register and will be passed by reference, adding code overhead.

Very few non-scientific applications use the type *long double*, so there was no use in making this representation any other than the same as for *float* and *double*.[6]

[3] Basically, if the computer controls anything with analog input or output and/or has hard timing constraints, it is a control-system, and normally has a critical regulator part in which floating-point calculations are important. Other systems which control a computer accessory do not normally need floating point calculations to that extent.

[4] This is true for Ghostscript version 5.10 and Perl version 5.004, which would otherwise have been good test-programs.

[5] See [Stal 92, "C Dialect Options"].

[6] This may change, as the need for a IEEE-754 64-bit entity increases if (or really: when) a Java virtual-machine will run on CRIS, since it needs a IEEE-754 64-bit floating-point type.

4.2.2 Non-fundamental types

The choice of representation for structures, bit-fields and unions has an extra dimension: padding — extra space between the members — can be present. This is probably the most common portability issue that *C* programmers are faced with: A known application-specific data structure is stored in contiguous bytes in memory. A pointer to a straightforward corresponding `struct`-definition is then directly mapped (type-cast) to those bytes. On a machine with member-wise padding (to where the application is then ported, often much later), the results will be surprising, and the program must be modified to a large extent. ANSI or K&R *C* says that any presence of padding between members of a structure or union is implementation-dependent, but this is often ignored if there is no such padding on the machine where the program was originally coded.

The CRIS architecture has no requirement for padding; when accessing a multi-byte entity on an odd address by word-size,[7] an extra cycle is needed. This is not a big penalty compared to the portability issue and the importance of compactness of data and code, so there is no padding of structure members in the CRIS ABI.

Likewise for unions; no extra padding, they are always the size of their largest member.

Bit-fields start where the previous bit-field left off, if any (i.e. no padding in-between), and extends from lowest numbered bit to higher bits. A bit-field of size zero means that the next bit-field will start on a new byte boundary. Non-bit-field members between bit-field members always imply padding to the next byte boundary.

4.2.3 Memory layout

The location and alignment of variables and constants is a grey area; it is language-dependent and dependent on the linker and assembler. However, the following can be said about where individual objects and functions will end up for CRIS: Constant objects are not modified, and it is desirable to keep them only in read-only storage, i.e. as the program code. A function *must* begin on an even address. Other than that, it is generally desirable for individual objects to start on 16-bit boundaries, so that `memcpy()` and other bulk accesses will be as fast as possible. Therefore, this alignment was made the default for constants, data or stack variables,[8] but modifiable with a compiler command-line switch.[9] However, there is no specific requirement or promise in the CRIS ABI for the alignment of any specific data.

[7] The ETRAX chip in which CRIS is implemented can work on the data bus in a 16-bit-mode or a 8-bit-mode. In 8-bit mode, no extra cycles are needed for odd-address word-accesses.

[8] Actually, only the size of the stack frame is adjusted. To align individual stack variables without other consequences, requires modifications to core parts of GCC.

[9] See page 24 and page 63.

4.2.4 Parameter passing

Previous informal measurements on intended target application code, made at the time of the earlier port attempt, showed that most functions have no more than four parameters. Thus it would be profitable to hold the first four parameters in registers.[10]

Parameters that are too large to hold in a register has to be passed by reference, pointing to a value located on the stack. All parameters take up the size of a register, even if passed on stack, with no promise on the contents of the unused part.

All parameters from the fifth, including references as described above, are put on the stack at increasing addresses.

4.2.5 Register usage

Since registers are faster than memory, often-used values should be held in registers. Such values are often "incoming" function parameters, local variables or other common expressions as found by the compiler.

Some registers are used for parameter passing, and the rest of the registers are left to GCC to take care of, for holding local and temporary variables that could end up in registers for performance reasons. The *C* `register` variable-qualifier has no effect when GCC optimizes.

The first assumption was to have `r0` ... `r3` holding the first four parameters, in increasing order counting from left in the source code. The parameter-passing registers should not be assumed to hold the original values after the call. It is not useful in the general case, and therefore not optimal to save them.[11]

It soon showed that the saving of the must-save registers, needing at least one instruction each, took no small amount of the function prologue code, and that this was not an optimal situation. There is a special instruction for saving multiple registers, the `movem` instruction. It has the restriction that it saves all the registers from `r0` up to and including the register specified in the operand; a "first register" or a non-contiguous set of registers can not be specified. So, the parameter registers, being assumed overwritten or *clobbered* at calls, were moved to `r10` ... `r13`, at the opposite end of the available register range.

Functions returning structures, must have the address of the return-value area passed to them at the call, if structure-returning calls should stand a chance to be re-entrant — a dedicated static per-function area is not sufficient.[12] Register `r9` was chosen to hold that address. It does not hold any value at the return from a call, so for all other functions it is completely free to be used as a scratch register. Finally, `r8` would be used as a frame-pointer for functions needing one, or as a save-upon-need register for all other functions.

[10] These static measurements were approximated to be valid for dynamic execution; see chapter 5.

[11] An assumption; see chapter 5 for how well it compares to actual measurements.

[12] See page 36.

It may seem like a better choice to make r0 the frame-pointer-register rather than r8, as that would always avoid a gap in the saved-registers list, so movem can always be used. I tried that, but it did not work for the following reason: The frame-pointer elimination pass is placed rather late in the compilation process. A register chosen as the frame-pointer register, but which can be used for other purposes, is *not* relieved of that duty until it is too late for ordinary temporary values and variables to go into it. This caused r0 to be unused most of the times. Therefore, it was really better to use r8 as the frame-pointer, and state a preference to use the other registers in order by increasing number before choosing r8.

4.2.6 Return values

Return values are best kept in the same register as the first parameter-passing register. This is because return-values from a function are often worked upon by the caller, and passed on in a call to another function, often as the first parameter. Then that result may be modified to form the return value from the first function. If the return value had been in another, non-parameter register, the passing-on would have had to involve a register-move operation. Example:

```
extern int foobar;
extern int baz(int);
int foo(int bar)
{
  return baz(bar+43)+foobar;
}
```

This results in the following code, when the first-operand register is the same as the return-value-register:

```
_foo:
        move srp,[sp=sp-4]
        addq 43,r10
        jsr _baz
        add.d [_foobar],r10
        jump [sp+]
```

It is left as an exercise to the reader, to find out the corresponding sequence that would have been the result if the return-value register had not been the same as the first-operand register.

4.2.7 The function stack frame

This is the most tricky part of an ABI. Great care should be taken to make sure the stack frame is optimal, in terms of space and speed for setting it up and destroying it for the most common case. A good measurement is that it should take no instructions at all to set it up and take it down, if the function has few or no parameters, and few or no local variables. This is easy to study in static, non-running code; a *lot* easier than figuring out the optimal register allocation.

The first choice is whether the stack should grow downwards towards lower addresses, or upwards towards higher addresses. The historical and easiest (at least when it comes to the public mind-set) is to let it grow downwards.

The stack-frame is set up in the *function prologue*. This is the code immediately at the function entry. There is also a *function epilogue* which takes care of de-allocation and register-value restore. The epilogue may however be located at multiple return-points in a function. This is profitable if it consists of no more than two short instructions.

Functions with a variable number number of arguments make the setup of the stack-frame tricky, because in that case, the unnamed parameters (those represented by "...") must be accessible in roughly the same way as an array. That is a problem if you keep some variables in registers, and the rest somewhere else. You need to move all the unnamed parameters in registers to a common "somewhere else". To be safe, this should work even if there is no function prototype present, i.e. when the caller does not know that the function has a variable number of parameters. Then the calling code can look the same, independent of how the called function sees the incoming data.

The easiest way to do this in gcc-cris was to find out if the function being compiled has a variable number of arguments, and then store possible unnamed parameters from registers before doing anything else in the called function (i.e. before storing the return-address), making the locations linear to the rest of any unnamed parameters.[13]

4.3 The porting

The ABI was at this time far from as clear as described above, but now at least I had figured out the contours, so I went on with the porting.

From the beginning of the coding of the port, to the point where it was possible to compile the compiler successfully, there was a long period where I had to work "blindfolded", without a chance to test it incrementally.

I started with the most simple instructions and no parameter-passing in registers, and added features as I saw opportunities in the compiled code.

4.3.1 The `tm.h` file

Lots of the initial work went into this file, to describe the architecture and ABI up to the point where it seemed enough to make a compilable compiler for CRIS.

I wasn't quite aware of exactly which target-specifying macros in the `tm.h` file were needed. Many of them have adequate defaults, but it wasn't obvious enough from the documentation which ones *had* defaults, let alone appropriate defaults.

[13] This solution was snatched from the ARM port.

I tried to follow the approach to use [Stal 92, “Target Macros”] as a checklist, defining or leaving undefined the macros in each of the subsections at the first pass.

As work progressed with writing the `tm.h` and `md` files, I made new passes over it, defining areas left with a default. Some of the macros provided wonderful opportunities to get stuck fiddling with.

Driver This was left to the last finishing touch; I used the preprocessor on the host system, until the final touch.

Run-time Target This could have been left undefined for long, if it had not been such a good place to put flag-definitions to control for example debug-printouts, through the definition of `TARGET_SWITCHES` et al. For CRIS, the normally used target-specific command-line-switches guide the generated alignment of data, as in `-m8bit`, `-m16bit` and `-m32bit`.

Storage Layout The majority of macros here are important, but are intermixed with the less important ones; those that were not needed for correct code. I staggered around here for a while, fiddling with optimizations.

Type Layout These macros were almost all straightforward, once I made up my mind about what type had what size.

Registers Most of these macros were determined by the architecture. The rest were interesting, for optimizing register usage across function calls.

Register Classes Also some straightforward macros. The constraint-defining macros were defined as needed, when I wrote the `md` file.

Stack and Calling These macros were easy to get caught up by, when writing the function-call-convention description.

Varargs This bunch were unimportant when compiling simple programs (without `printf()` of course). The only tricky bit was to put the unnamed values in the right place, but when using a solution from another port,[14] this became pretty easy.

Trampolines This part was quickly hacked up and then left alone, and was even uncompilable for a long time due to a syntax error in the in-line assembler code, and was still recently buggy.[15] The only way to make use of this functionality from a *C*-program, is to use the GCC extension of nested functions, and taking a pointer to a nested function and calling it in another function.

Library Calls All these macros had obvious definitions. For most of them, it was the default value.

[14] See page 62.

[15] The static-chain register was not saved before modified in the trampoline, but the caller assumes it to be call-saved.

Addressing Modes This was pretty straightforward, except for controlling the bugs that are so easy to put into `GO_IF_LEGITIMATE_ADDRESS` and its helper macros. It was during these debugging sessions that the benefit of having a dword-variant of register-plus-index addressing[16] became apparent.

Condition Code For CRIS, having a condition-code register, no other macros than `NOTICE_UPDATE_CC` had to be defined here. To diagnose bugs that crept into the definition of that macro, I defined a compiler flag `-mcc-init` to turn off the use of the condition-code register result of earlier operations.

Costs Nothing needed to be defined here for the first porting rounds, and is still much unexplored, as the first-attempt execution and code costs look fine.

Sections After all recommendations from [Stal 92] had been followed, there was nothing more to define here.

PIC No definitions were applicable for CRIS.

Assembler Format No surprises here either.

Debugging Info For the primary purpose of just building a compiler, this section was irrelevant. I stole everything from the Sun 3-definitions.[17]

Cross-compilation I got away with not defining anything here, by always assuming that the host uses the IEEE-754 floating-point format too. Besides, there was a fatal bug related to floating-point interpretation in the GNU assembler that was used. The easiest workaround (and actually faster than interpretation) was to just output the actual binary representation of the IEEE-754 value as an integer, and use that as floating-point data. For example, `1.0` became `0x3f800000`. Since no serious floating-point work was projected, these assumptions seemed to make sense.

Misc All of these were trivial, maybe except for size of the elements in a `switch { case ...: }`-table.[18]

4.3.2 The `md` file

Instruction patterns

Just as with the `tm.h` file, I followed [Stal 92, "Standard Names"] and looked for named patterns matching CRIS instructions.

A major decision was whether to describe the side-effect addressing-mode presented on page 16. There were doubts that this was expressible at all to

[16] See page 13.
[17] It showed later that this worked out-of-the-box when used by a port of `gdb`.
[18] See page 67.

GCC, much less beneficial. As it turned out later, this is indeed used, although it complicates the machine description with one extra instruction pattern for each instruction where this addressing-mode is applicable. There is one instruction type where this is not expressible to GCC: instructions using condition-codes. For CRIS, this is `test.`*S* `[`*side-effect-expression*`]`. Extra move-instructions introduced during the register allocation phase will cause the side-effect to be moved out of the instruction, and then interfere with the state of the condition codes. When this happens, the situation is detected by GCC and there is a prompt call to `abort()`.

The sign- and zero-extend variants of load-instructions and some of the arithmetic instructions were relatively easy to specify, in comparison.

Splitting

To fill delay-slots more effectively, I wrote a few `define_split` patterns. It seemed like there were no big opportunities for splitting; some sub-optimal side-effect- and three-operand-patterns that could appear after register allocation were taken care of.

After studying some resulting assembler code, I came to think that instructions such as `move.`*S* `[r`*X* `+ constant_index],r`*Y* can be split at no cost for most values of `constant_index` and provided that `r`*X* is not used after this instruction, until it is assigned a new value[19] (i.e. it is "dead"). The resulting instructions would be `add.d constant_index,r`*X*, and `move.`*S* `[r`*X*`],r`*Y*. The `move`-instruction could then always be put into a delay-slot, and the `add`-instruction too, for values $-64 \leq$ `constant_index` ≤ 63. The `add` instruction will automatically be matched against the optimal pattern for a known value of `constant_index`.

On the surface, this seemed like a reasonably sane optimization; there are "notes" attached to the RTX-representation that hold such information. This specific type is called `REG_DEAD`-notes. I was a little suspicious over the placement in [Stal 92] of `PRESERVE_DEATH_INFO_REGNO_P()` in the *Obsolete Register Macros* section, and the negative description of its mandatory definition when `REG_DEAD`-notes were to be used, but decided to give it a try. This was wasted time; the `REG_DEAD`-notes were still not correct in all cases when present, regardless of `PRESERVE_DEATH_INFO_REGNO_P()`.[20] Naturally, this caused hard-to-find bugs. It shows that when the register allocation pass is finished, these notes hold bogus information, and that GCC would have a hard time to keep these notes up-to-date after that point.

[19]The case where X equals Y is taken care of by other splitting patterns.

[20]This has been changed in egcs version 1.0.2 and gcc version 2.8.1. The function used to find them, `dead_or_set_p()`, should no longer return notes after when they may have become incorrect.

Attributes and delay-slots

The use of an attribute as `slottable` (see chapter 3.2.4) to help filling delay-slots, was described in [Stal 92, "Delay Slots"]. It was not clear at the time whether condition-code tracking would have been improved by using attributes. Maybe it would.

The methods by which to describe various incarnations of delay-slots were pretty well described, but not really how to output a `nop` when the delay-slot was not filled. Thanks to the SPARC port, I found out that operand punctuation could be used to tell `PRINT_OPERAND()` when and where to output the stored delay-slot-fillers, or a `nop` if the delay-slot was not filled.

This is a typical example; whenever there was an issue with a specific detail in the machine description, I found that the CRIS architecture was generic enough that each specific feature and its implementation-problems had been taken care of in some other port. Unfortunately GCC 2.1 still had many problems that appeared with the CRIS-specific *combination* of these features, so a lot of time went into debugging the GCC core instead of the port, just to find that the bugs I found had already been fixed, and would be in the next release.[21]

Peephole optimizations

These were added whenever ugly code was spotted in the resulting assembler code. Even though there are only 16 of these patterns, they caused quite a few problems, with condition-code tracking and errors in the patterns. Maybe it would have been better to leave them out; the sub-optimalities they handle are mostly random spottings.

4.3.3 The `tm.c` file

The `tm.c` file is really nothing more than an extension to the `tm.h` file and somewhat to the `md` file. No development phases were related specifically to it. Whenever a macro in the `tm.h`-file was obviously better implemented as a function, it was added here.

Some relatively labor-intensive functions were `function_prologue()`, `function_epilogue()` and `notice_update_cc()`. They simply have an unexpected lot of combinations for their input and context. When updating the condition-codes, at least one new case had to be added whenever an optimization was attempted somewhere else, but it seemed that there was always another case that was overlooked. Care had to be taken to the cause of the current condition code, the instruction specifics, and its operand combinations. For example, some of the operand and instruction combinations are expressed as shorter equivalents, but which do not set the condition codes as expected by just inspecting the RTX representation.

In the prologue/epilogue case, each function come in various colors: The stack frame layout varies, as well as the register- and parameter-needs. My

[21] There is better access to GCC-development information than I knew then, see appendix C.

attempts to optimize the instruction-sequence for stack-pointer adjustment and register-allocation and register-restoring often escaped the bounds of correctness...

4.3.4 Language-specific features

Before reading [Stal 92], one of my initial assumptions was that there would be lots of language-specific constructs with specific translation machinery to take care of. Though my worries were a bit exaggerated, there are a couple of optional machine-specific patterns that apply to, for instance, the standard `strlen() and strcmp()` functions.

These are the ones that were interesting for the CRIS port:

Switch/case

This construct was very common in the intended application,[22] so if any specific corner of the compiler was to be scrutinized, this was a good candidate.

There were two choices for describing the meat in a `switch { case ...: }` construct, i.e. how to conditionally jump using a index to a table. There is a choice between specifying a simple jump-using-index and a full-blown case-jump-instruction. In both cases, the table is an array, corresponding to continuously increasing "cases". For the simpler jump, the index into the table is already checked for upper and lower bound. The density of the array is controlled by the GCC core, which inserts binary-search branches for sparse case-ranges. There is support for the port to control the density through the macro `CASE_VALUES_THRESHOLD`, but the default is good enough for CRIS.

The entries in the array, can be either the absolute address to jump to, or a value relative to the start of the table. Clearly, using dword-addresses in the table would be overkill. It is more compact to make the values relative and word-sized. This would suffice for most case-tables, but I was worried about the risk for overflow in the entries, causing assembler errors or silently wrong code. While this is far-fetched when ordinary human-written code is compiled, it is much more probable for machine-generated code. The total amount of the code at the "cases" just has to be big enough for *some* entry to overflow. I took the dword and simple-jump choice in the beginning. After consultations with the author of the GNU *gas* assembler port and the senior architect of CRIS, my advisor, there was apparently minimal risk in using relative values. The benefit is clear: a case-construct with word-size relative entries is more compact than the corresponding dword table-jump, and can be faster if word-size data is sufficiently easily handled. The assembler can in most cases correct "overflowed entries",[23] or will in rare cases emit an error, which is satisfactory.

[22] See chapter 4.6.1

[23] If the table is sufficiently small (less than approximately 4K entries), but the relative addresses to the code for each case do not fit, then stubs with `jump` instructions are inserted just after the table. This is called *broken word* handling.

GCC has a few shortcomings that stops it from automatically finding the optimal code sequence itself; it knows about, and *could* profit from using an unsigned-minimum instruction such as `bound` to check the index range for a table-jump, but it doesn't. Also there is no possibility to recognize the program counter as a register, so it could not use normal addressing-modes and an `add` instruction to perform the jump; it needs a specific table-jump instruction.

Therefore, the more complex `casesi` pattern is expressed as a `define_expand` to the `bound` instruction and `adds.w [pc+rX.w],pc` for the optimal instruction sequence.

For example, the following code snippet:

```
int j;
switch (i)
{
  case 2:
    bar();
    j = 4;
    break;
  case 3:
    baz();
    j = 2;
    break;
  case 5:
    foobaz();
  case 6:
    foo();
    j = 1;
}
```

will compile to (`i` in `r10`, `j` in `r9`; indentation and labels have been enhanced):[24]

```
   subq 2,r10
   bound.b 5,r10
   adds.w [pc+r10.w],pc
table:
   .word case_2  - table
   .word case_3  - table
   .word default - table
   .word case_5  - table
   .word case_6  - table
   .word default - table
case_2:
   jsr _bar
   ba switch_end
   moveq 4,r9

case_3:
```

[24]The experienced reader will see that this is not really an optimal code sequence, at least with respect to code size. Well, it is just an example.

```
   jsr _baz
   ba switch_end
   moveq 2,r9

case_5:
   jsr _foobaz
case_6:
   jsr _foo
   moveq 1,r9
default:
switch_end:
```

By checking the index with the `bound` instruction, the upper bound will be substituted for values higher than it; an unsigned-maximum operation. The index has to be normalized to zero first, by subtracting the first index value. Because of the nature of twos-complement-representation, any resulting negative value will appear as a very large unsigned value, and so be substituted with the upper bound. By using the upper-bound value as a `default`-case, this is handled by just adding an extra entry to the end of the table. GCC specifies the location for the `default`-code to the `casesi` pattern, or the end of the switch when there is no default.

In order to describe the `casesi` sequence with `pc` as an ordinary register in the `adds.w` instruction, I had to cheat a little. The expansion pattern for `casesi` explicitly mentions register `r15` which indeed *is* an alias for `pc`, but GCC does not understand that; it causes confusion to the data and control flow analysis. To work around this, I had to add dummy "move" instructions, moving `r15` to and from `pc`, and matching patterns with no assembler output. Precautions were also needed to avoid that other instructions crept in between the real "jump" and the dummy "move", since reload would otherwise move instructions here in some rare cases. Therefore a `use` of the indexed register was added to the dummy "jump" to avoid instructions modifying that register, and also a definition of `FINAL_PRESCAN_INSN()` that moves mis-located instructions to before the real "jump".

An alternative would have been to add a specific pattern for `adds.w [pc+rX.w],pc` but I considered that to be cheating even more with the machine description: This way, there is a chance for the `adds.w` to be combined with, or split into other instructions.

Block copy

The ability to copy memory blocks larger than the largest simple type is present in most languages. It is visible to the *C*-programmer as using a single assignment to copy structures, or passing structures to and from functions by-value. Calls to `memcpy()` can be identified and intercepted by GCC, and actual calls to this function are generated when there is no better method in sight for GCC.

GCC takes it all the way down to the instruction level: There is an instruction pattern named `movstr` for this purpose. It is not beneficial to define this pattern for CRIS, as GCC will emit an optimal sequence of move instructions for small moves of known sizes.[25]

You can control the number of instructions that may be emitted, through the macro `MOVE_RATIO`. It specifies the number of memory-to-memory sequences, *below* which inline move instructions are emitted. Note that this is not the actual number of instructions; for an architecture such as CRIS that does not have a direct memory-to-memory instruction, it is the number of instruction pairs to move a datum between memory positions.[26] The default is 15, which would mean that a maximum of 28 instructions, moving 56 bytes, would be emitted.

Because the intended target would be code-size sensitive as well as speed-sensitive, I decided that the threshold would be set to allow 32 bytes in a move, i.e. `MOVE_RATIO` is set to 9. This decision is based on ad-hoc reasoning — the inline `memcpy()` for moving 32 bytes, takes sixteen words of instructions. There would be a penalty of approximately eight words for a call to the `memcpy()` function: The call instruction, most often worth three words; the `moveq` *`size`*`,r12` instruction (for the third argument to `memcpy()`) and some uncertain penalties, probably worth at least four words of code on average. These less tangible penalties depend on the other code in the function. Examples are the cost of possibly ruining the leaf-function-ness of a function, the comparative extra cost in register allocation, and the cost of moving registers around to the source and destination parameter registers. By setting the threshold to 32 rather than 16 bytes, speed is valued a bit higher.

4.4 Tools

These are the tools I used during the development of the CRIS port. They are not specific for use with GCC, and the GNU tools are not even specific for Unix systems — anymore. However, naturally other GNU tools besides GCC are very useful when you write and debug a GCC port.

4.4.1 Editor

The editor environment *emacs* was especially useful, as I could have the on-line hypertext version (known as the *info* format) of [Stal 92] available in one buffer, while working on GCC source in another window. Support for compilation with any external compiler-type program, together with a feature for quick lookup of compile-time errors, are part of the emacs package. There's also an interface for running a debugger in yet another buffer, with automatic tracking of the current location, in buffers with the source code.

[25]There *has* been some unfortunate bugs before version 2.7.2 that made it better to define such a pattern than leaving it to GCC.

[26]To *not* accept a memory-to-memory move in the predicates for the move-patterns, will result in a fatal error. As far as for version 2.7.2, the *constraints* must be used to split it up in two instructions.

4.4.2 Debugger

Speaking of debugging, the GNU debugger *gdb* has as expected good features for being run in a buffer in *emacs* and then some valuable breakpoint features. The GCC package comes with a few macros for use with GDB, packaged in a `.gdbinit` file which will automatically be read in by GDB, when a debugging session for a compiler part is started.

4.4.3 Compilation management

A variant of the *make* program is necessary for compiling GCC. Fortunately, the GCC installation creates a project file, the `Makefile`, in a format which is compatible with most variants of make, including the *make* from SUN microsystems and, of course, the GNU *gmake*.

4.4.4 Compiler

It was natural to compile GCC with another variant of GCC, this time the host compiler. From time to other, I used the "native" Sun `cc` (this was SunOS 4.1, so the compiler was still included with the operating system) to check that the port-specific code was portable and still compilable with another compiler, and that no ANSI-specific features had crept into it.

4.4.5 Debugging measures built into GCC

Dumps from the different compilation passes are available, with different "`-d`"-switches, or preferably, "`-da`" for them all. This dumps the RTL representation of the internal state after each pass into different files, which was invaluable for getting a grip on where to start debugging. The output carries port-specific information, such as the name or relative position of the matching pattern and register names instead of just numbers.

4.5 Debugging the port

Every unexpected situation that is detected in the GCC core results in an `abort()` call. This makes bugs in a port either manifest themselves as invalid code or as fatal "signal 6" (abort) errors; seldom as the elsewhere more common *illegal memory access*[27] or just plain "hanging".

For the work on the CRIS port, this meant that as each pass uses the machine description in a different way,[28] any incorrect code caused by bugs in the port, was likely to be introduced in one specific pass, after the basic representation had been generated. If there was a problem with the basic RTL, then it was due to an error in the machine description macros in the `cris.h` file, or a faulty `define_expand`-pattern.

[27] This behavior has many names; "General Protection Error" SIGSEGV, you name it...

[28] See [Stal 92, "Passes"] or chapter 3.2.6.

The RTL dumps described above, were then used to backtrack from the point where a bug was spotted (whether it was incorrect code or an `abort()` for a non-obvious bug), through the dump-files in opposite chronological order, until the RTL representation looked correct.

After I found the immediate pass after which the bug appeared in the RTL, I set a breakpoint in the main loop over instructions in that pass, for the *insn* with the corresponding number (*insn uid*) for where the buggy code was seen. Then I stepped onward through the rest of the pass, until the cause of the bug was spotted.

Other times I did a binary search through one or several passes, narrowing down the places before and after where buggy code had been generated. This was not as hard as it sounds, with suitable use of breakpoint-conditions in GDB.

4.6 Testing the port

I can not enough emphasize the importance of a working test-bench system with time-measuring capabilities. There is currently no better way you can make sure that a small modification to the compiler does not in fact cause worse code for the whole application, or even worse, incorrect code in some cases. Also, running *real* programs is needed for testing; it is hard to identify optimal code from a compiler by just looking at small artificial test examples in *C*, and checking the resulting assembler code. You may spot some grave errors and performance problems, but you cannot see the more subtle bugs that will come and bite you when the compiler is used in production.

A suitable test environment must have a simulator (or actual system) with some kind of simple cycle-true measurement capabilities, and a program with *filter* characteristics, performing work similar to, or the same, as the intended target system. A filter program is any program that simply takes a fixed input and produces a fixed output, regardless of external events during the "filtering". As such, there is no need for any input/output operations except basic file operations, for example those in [ANSI C]. Don't forget that there should be enough input to keep the program running for a measurable time, and that the input should be close to the typical real-world case.

Using this type of test environment, you can accomplish two things:

- Make reasonably sure that the compiler outputs code that gives correct output and the best possible performance.
- Create an automatic test machinery that compares the time and output to that of previous runs.

The test machinery is needed e.g. whenever you have to modify the target-specific code of the compiler, or when testing for new versions of the target-independent part of the compiler. This type of test is called a regression test.

4.6.1 IPPS

The original usage of CRIS was aimed at data conversion at multiple levels, such as network protocol operations and presentation format conversions. One of these format conversions was from the IBM graphics description format IPDS to the more commonly used PostScript printer language. There was already an existing program called IPPS, which used code from the converter products, taking input in a home-brew file format. The available input included artificial IPDS test patterns and actual recordings of typical print-jobs. Thanks to the modularity of this program and the large amount of test cases, this program made a great test case for the compiler (and the target architecture).

4.6.2 GCC itself

You may object that GCC[29] is not intended to run on the target system. That is correct, but its availability and access to appropriate input test-data (again the GCC compiler!) made it a good case for a different viewpoint to that of the IPPS test case; these programs have big differences in code style. IPPS has small simple functions and different data sizes (byte, word and dword, with a focus on byte and word), with operations often including two sizes.[30] GCC has large complicated functions and mainly `char`s, `int`s and pointers (byte and dword sizes, with a focus on dword), with operations mostly on the same size. These differences sum up to a small, but still measurable variation in instruction and data type usage.

As a general reflection, I believe that the magnitude of these differences is not probable to exceed that of differences within different real-world systems using the CRIS architecture;[31] if a modification to the port results in better code for these two cases, it is likely to be beneficial for other code as well.

[29] The version of GCC used as test-program and input is actually version 2.1. For the sole purpose of comparative and regression tests of *later* versions, there is no point in "upgrading" this to a later version.

[30] This is the main excuse for sign- and zero-extend "built-in" into arithmetic instructions.

[31] It may be that this only applies to programs written in *C*. Although very similar to *C*, compiled *C++* code has a tendency towards smaller function and more frequent use of indirection, most notably for function calls, resulting from use of virtual functions.

Chapter 5

Measuring gcc-cris

With a simulator, a set of test programs and a not-yet-fixed architecture, there is opportunity to get carried away and measure any architecture- or compiler-related quantity, such as the optimal number of registers or the effect of different addressing-modes and instruction types. As the purpose of this work is a port of GCC to CRIS, the measured object would be a fixed architecture, restricting variables to the port specification only.

5.1 What to measure

Theoretically, the best possible port should describe all features and addressing-modes to GCC. If any of these descriptions is not beneficial for the generated code, then it's a bug and shortcoming of GCC, and should be corrected in the GCC core. I therefore assume that there is no point for the purpose of a good port and machine description, to measure any quantity that relates to the level of detail in the machine description: Any measured beneficial effects of low detail in the machine description should hopefully go away in future GCC versions. So there should be little point in detailed experimentation with the `..._COSTS`-macros and whether to include some instructions or addressing modes in the port — that should all be subject to trivial reasoning for optimality.

There are a few general dark corners in GCC, where the result of choices is documented as being uncertain for performance; for example the effects of the flag `-fforce-addr`. I believe that these effects will probably vary with each GCC version, but as optimizations are put in and bugs removed, the outcome will eventually be clear and fixed for the major architecture types.

For CRIS with no fixed ABI, there is still a big port-specific ambiguity: How good was the choice of register usage for the fourteen general registers? Would another choice, close or far away, be better? I believe this was something that needed specific investigation. There may be other areas that could benefit from closer measurement, but it is less obvious that they are not covered by previous arguments in this report.

5.2 Register usage unknowns

For a GCC port, the register usage description has two big dials. One is how many registers are used for parameter passing, and the other is how many registers may be overwritten, or *clobbered* at the call. They will be measured independently. Register allocation for special values and register allocation order are potentially other variables, but for fixed values of the other two, they can be figured out for CRIS:

GCC allocates registers for temporary values with short lifetimes first, so the registers that are call-clobbered should go first. GCC knows that, if possible, it should not allocate a clobbered register to a value that must be available *after* a function call. Among the clobbered registers, allocation should be in reverse order of likelihood of being used as a parameter. The saved registers should be allocated in order from `r0` and up, so they can be saved with a `movem` instruction. In any event, the register used as the frame-pointer should be the last saved register, since it holds up a register until late in register allocation. Other sometimes-special-purpose registers should be allocated where they affect performance as little as possible, if the special purpose does not apply or is not used for that particular function. Allocation of registers for special incoming values, will have to avoid collision with possible parameter registers.

There *is* another related unknown: Parameters passed in registers should be stored in either saved or clobbered registers. Situations can be imagined where either method could be profitable, and neither should be ruled out without measuring.

For example, a parameter should maybe be put in call-clobbered registers, because that means that the caller will not have to save that register in order to call. Then the called function can modify the register at will, without saving it or moving it to a clobbered register. This assumption was followed for the current ABI.

On the other hand, a parameter-passing register should maybe be saved, because the caller may use that value after the call. This will happen if the called function is in a loop, and the loop-counter is passed as a parameter. (I admit that this is subject to assumptions, such as that the calling function needs that specific parameter value after the call, and that other circumstances makes it preferable to keep that value in the same register as the one used for the parameter.)

5.3 Domain of measurements

With the method of storing arguments in a pre-allocated area instead of pushing them on stack,[1] GCC sets limits such that you *must* have enough registers to hold parameters for library functions, and you cannot pass them by reference. As this means two registers for each `long long` parameter, and two parame-

[1] See page 34.

ters for e.g. multiplication, you would have to have at least four registers for parameters, if you need arithmetic for `long long`.

Fortunately, neither of GCC or IPPS use `long long` types for arithmetic. I was therefore able to get away with three registers for parameters, for the purpose of these tests, as calls to library functions do not need more than three parameters.[2]

I see no more profitable place to put the return value than a register. The return value must be able to take up two registers, for the sake of library functions returning `long long` values. I could have crippled GCC to *not* compile the library routines that return `long long` values, but I believed that results would show that the direction of that few clobbered registers is not optimal anyway.

The bidding therefore starts with parameters in *three* registers, and *two* clobbered registers, for the return value. The upper limit for parameter registers depend on the number of function-dependent special-purpose values passed in registers. There are three of them: the structure-return-address, the frame-pointer and the static chain.[3] The frame-pointer could possibly be used as a parameter-passing register, but the calling function would then have to save the register around the call, since it would need the frame-pointer after the call, and the called function may need the frame-pointer as well, resulting in more overhead instructions. For the same reason, the frame-pointer must be in a call-saved register.

This means there can be at most thirteen call-clobbered registers, and eleven parameters in registers.

There are of course "degenerate cases" with both saved and clobbered parameter approaches, where parameters end up in *both* saved and clobbered registers. While such a mix could theoretically be consciously used and even optimized, the effects would probably be application-specific and no such attempts are tried here. These degenerate cases will be included in the measurements, because it was easy and for sake of completeness.

5.4 Measured functions

This leaves us with four related functions:

$s_c(p, c, f)$ The size of a compiled program when parameter-passing registers should be call-clobbered.

$t_c(p, c, f_i)$ The execution time for the program as above.

$s_s(p, c, f)$ The size of a compiled program when parameter-passing registers should be call-saved.

$t_s(p, c, f_i)$ The execution time for the program as above.

[2] GCC considers *some* library functions as needing special parameter-passing. It is not the *whole* of the ANSI *C* library, only those functions that could be real instructions, such as mathematic functions, for example `sqrt()` and memory-block-functions such as `memcpy()`.

[3] See [Stal 92, "Frame Registers"].

Here, c is the number of call-clobbered registers, p is the number of parameter-passing registers, f is the filter program[4] and f_i the filter program when running with input i.

The f will be IPPS and GCC, with i being a real-life recording of typical input for IPPS, and `combine.c`, the biggest and meanest *C*-file in GCC, for input to GCC.[5]

The domain defined by the constraints above, is $3 \leq p \leq 11$, $2 \leq c \leq 13$. (The degenerate cases, where parameter-registers end up in both saved and clobbered registers are for call-saved parameters $p + c > 14$ and call-clobbered parameters $p > c$.)

The compiler flags are set to the normal production-code optimization level of `-O2`, which should give an optimal speed and size with no automatically inlined functions.[6]

5.4.1 Parameter-passing formula

It is essential that we get as good as possible register usage for parameters, for every c and p. Therefore we need to establish an optimal correct parameter usage for saved and call-clobbered parameter registers here. For call-clobbered registers, the optimal usage chosen (there may be more than one), should preferably coincide with the current ABI, unless that formula is shown to be suboptimal. The call-saved usage will *not* coincide with the current ABI.

Clobbered parameter registers

For the call-clobbered alternative, I will try to keep the last clobbered parameter register in `r13`:

`r`*`first-parameter`* = foo (`r`*`first-parameter`*, …, `r13`[, `r0`, …]);

The degenerate cases for parameter registers will wrap around and continue from `r0`. The frame-pointer will be the highest numbered call-saved register.

Specials will preferably be in call-clobbered registers: the structure-return address in the register with the next number lower than `r`*`first-param-reg`*, and the next lower register will contain the static chain. If there are not enough call-clobbered registers, the one or two registers with lower numbers than frame-pointer will be used, with the static chain going first, and structure-return-address coming last. The first parameter register also contains the return-value.

For comparison, with $c = 5$ and $p = 4$ this matches the current parameter-passing scheme:

`r10` = foo (`r10`, `r11`, `r12`, `r13`);

[4] See chapter 4.6.

[5] Your mileage may vary; `insn-recog.c` may beat it for sufficiently complicated target architectures. Anyway, `insn-recog.c` is machine-generated and cannot be regarded as typical input because of its structure.

[6] In egcs 1.1 a new interesting optimization flag is added: `-Os`, for optimize-for-size, which gives the effect of `-O2` but avoids some possible code-enlarging optimizations.

Since there is one more call-clobbered register than the parameter registers, i.e. `r9`, it will hold any structure-return-address. Register `r8` will hold the frame-pointer and `r7` will link to any static chain.

Saved parameter registers

For the alternative with call-saved registers, parameter-passing registers start at `r0`, counting up:

`r12` = foo (`r0`, $\nearrow$, `r`*`frame-pointer - 1`*[, `r`*`first-call-clobbered`*, $\nearrow$]);

The degenerate cases where the number of parameters in registers are more than the number of saved registers, will need some attention to work correctly: The last saved register, the frame-pointer-register, will not carry a parameter. Instead, the higher-numbered parameters will continue at the lowest clobbered register.

Registers for special incoming values will then too, be preferably located in saved registers. If that is not optimal, then passing parameters in saved registers isn't optimal at all. The returned-structure-address will be in the next-to-last saved register, except when it is used as a parameter register; then it will be in `r13`. The static-chain-register will, when that register cannot not hold a parameter, be the next lower-numbered register from the returned-structure-address register. Otherwise, it will be `r13` if the returned-structure-address isn't there, else the first call-clobbered register that is not a parameter.

Registers `r12` (and theoretically `r13`) always contain the return value, and are therefore always clobbered.

5.4.2 Register allocation formula

Besides function parameters, we need to consider the order in which the rest of the registers are allocated for instructions, local variables and other temporary usage.

As mentioned on page 75, call-clobbered registers will go first, and saved registers last, starting with `r0`. The order among the call-clobbered registers depend on their usage:

Clobbered parameter registers

The preferred register allocation order will be:

{[when available: `r`*`lowest-call-clobbered`*, $\nearrow$, `r`*`first-param - 1`*,]
`r13`, `r12`, $\searrow$, `r`*`first-param`*, `r0`, $\nearrow$}

Just as for parameter-passing, this matches the current scheme for $c = 5$, $p = 4$, `r9` being the only available call-clobbered register before the parameter registers:

{ `r9`, `r13`, `r12`, `r11`, `r10`, `r0`, $\nearrow$}

Saved parameter registers

The register allocation order is:

{[when available: `r11`, $\searrow$. `r`*first-call-clobbered*,]
 `r13`, `r12`, `r0`, $\nearrow$.}

Because registers `r13` and `r12` may be used for return value and incoming structure-return-address, they are allocated last of any other call-clobbered registers. The order for other call-clobbered registers with no fixed purpose does not normally matter. For sake of consistency with the degenerate case when call-clobbered registers are also used to pass some parameters, they are allocated with decreasing register numbers.

Call-saved registers will be allocated from `r0` and up, so there is a potential clash with parameter-passing registers. However, since the stated register allocation order is used only secondary to other register preferences, it should not interfere with register allocation for incoming or outgoing parameter registers.

5.5 Practical limitations of the test

Some generalization of register usage parameters in the port were performed. I had to remove ABI-specific assumptions and optimizations in the library functions in order to create a working and equal environment for all combinations of c and p. Other than equalizing the execution time, these modifications should not affect the outcome. For comparison, the values for the current unmodified system is displayed.

Some boundary values of p and c were not accepted by the register allocation algorithm in GCC when compiling the test programs. This caused a fatal error with a message that should be well-known to GCC programmers:

```
fixed or forbidden register was spilled.
This may be due to a compiler bug or to impossible asm
statements or clauses.
```

This often happens for an architecture where `SMALL_REGISTER_CLASSES` should be defined but isn't.[7] I checked that these cases are indeed borderline cases of p and c and that the limitation was indeed due to register allocation restrictions and not a bug in the port description or an obvious GCC bug. After that, I deleted those entries with no further action. See table 5.1 for an overview.

5.6 Results

The eight sets of results show execution times in cycles and code size in bytes, for call-clobbered and call-saved parameter registers, for each of IPPS and GCC, while varying the number of registers used for parameters and the number of call-clobbered registers.

[7] See page 30.

p	c	Parameters in registers are	Fail point
11	2 ... 13	saved	IPPS (all in the same function)
9 ... 11	2 ... 13	saved	GCC reload.c : find_reloads()
10	12 ... 13	clobbered	GCC reload.c : find_reloads()
11	11	clobbered	GCC reload.c : find_reloads()
11	12 ... 13	clobbered	GCC loop.c : strength_reduce()

Table 5.1: Failing test-cases

These measurements are probably best visualized with a three-dimensional plot where the surface is the resulting number of cycles or bytes, with contour lines near the interesting points. The contour lines are computed as (non-discrete) linear, with contours close to the results of the p and c giving fastest results for the two programs, as well the result when using the current ABI ($p = 4$, $c = 5$) for call-clobbered parameter registers. Arrows mark the surface and contour value-points of those interesting pairs of p and c.

The cycle counts do not include the "extra" cycles needed when reading and writing "misaligned" memory data.[8] Since there is no way to tell GCC about the relative penalty or how to avoid it other than for statically located data,[9] the difference in this figure alone would add noise rather than information. (The "misalignment cycles" were slightly positively correlated to p, c and the execution time, but would *not* have affected the outcome. Compared to the "normal" cycles, they were in the order of 4% for IPPS and 1% for GCC.)

The size is for code only; the size of the static data was (naturally) the same for all p and c. See table 5.2 for key measurements.

Program (f)	p	c	$t_c(p, c, f_i)$, cycles	$s_c(p, c, f)$, bytes	Comment
GCC	4	5	2680058980	736564	ABI-specific "libc"[10]
GCC	4	5	2682757147	736504	
GCC	4	8	2727410222	755480	
GCC	6	5	2670776136	733252	
IPPS	4	5	4448778528	278820	ABI-specific "libc"[10]
IPPS	4	5	4460745789	278760	
IPPS	4	8	4313296711	280104	
IPPS	6	5	4459590764	278380	

Table 5.2: Key measurements (for call-clobbered registers)

[8] See page 59.

[9] That is, without other, more unpleasant side-effects, such as structure-element padding.

[10] Mostly effects of a slightly better ABI-optimized memcpy().

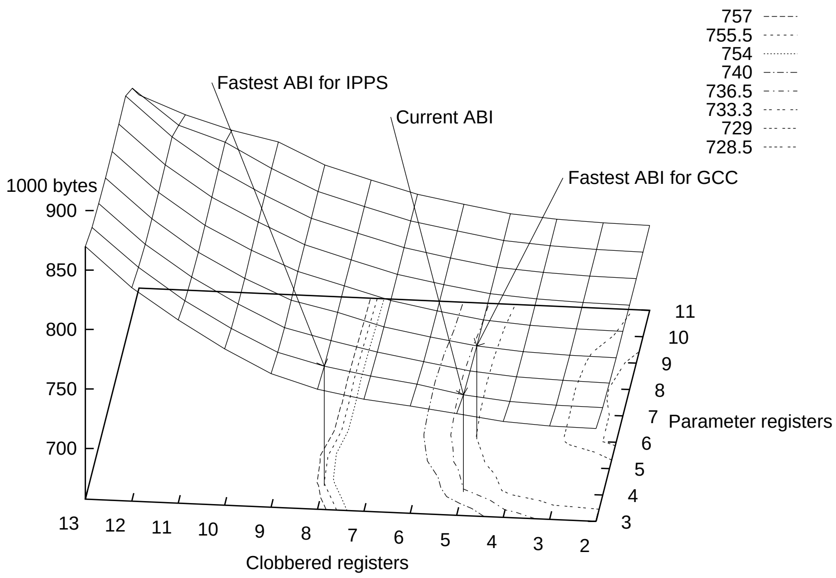

Figure 5.1: $s_c(p, c, \mathrm{GCC})$, size with *clobbered* parameter registers

5.7 Analysis

As can be seen by briefly comparing the figures (5.1 with 5.5, 5.2 with 5.6, 5.3 with 5.7, 5.4 with 5.8), having parameters in saved registers is an overall bad idea both for code size and performance, and so will be disregarded and not further analyzed in the following.[11]

5.7.1 Code size

For both GCC and IPPS, the curve shows a slope towards fewer call-clobbered registers, but the angle is very flat below eight registers for both GCC and IPPS. There is a very slight drift towards seven or eight parameter registers.

This comes as no a surprise; the `movem` instruction makes it as cheap (in terms of code-size) to save many registers, as it is to save one register, as long as all registers are continuous starting with register `r0`. Less call-clobbered registers also means less code needed to store and restore variables and temporary data to stack locations before and after calls.

[11] I *did* check that GCC actually did recognize the parameter registers as "saved" and made use of their previous contents after a call.

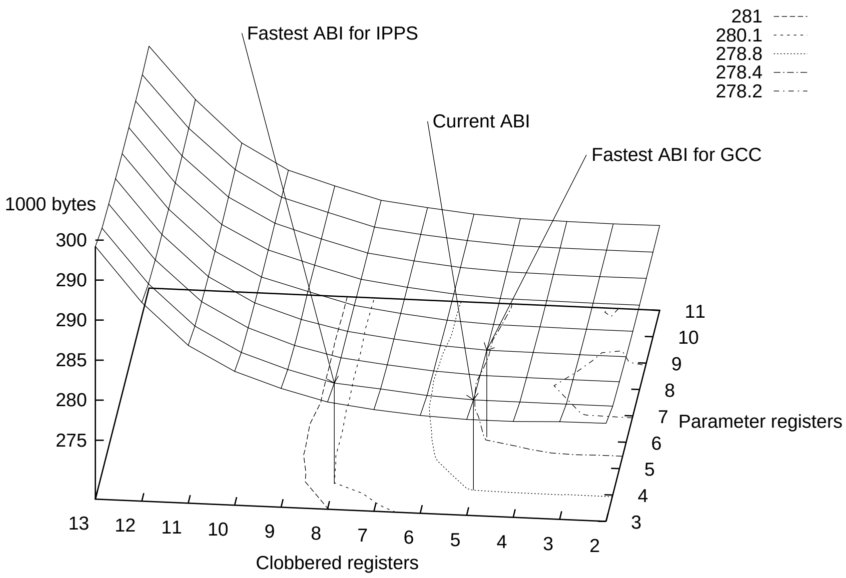

Figure 5.2: $s_c(p, c, \mathrm{IPPS})$, size with *clobbered* parameter registers

Why the number of parameter registers is so unimportant to code-size is not as obvious. It seems all effects even out, making the differences in code size only in the order of 1/10000 when varying the number of parameter registers for a fixed number of call-clobbered registers.

I believe the differences in code-sizes for the interesting points, for both applications at most 4%, are small enough to be secondary to the points of fastest execution.

5.7.2 Speed

The difference in speed between the optimal ABI's for these two applications are at most about 3%, although I would guessed them to be a lot smaller: less than 1%.

As for the code size, the number of parameter registers is much less important than the number of call-clobbered registers. Yet, the current number of parameter registers not surprisingly gives optimal speed for IPPS; the assumptions from static observations were right on this point. A program with the characteristics of GCC, with its optimal *six* parameter registers, suffers only slightly (the current ABI is 0.45% slower), so it would definitely be no point in

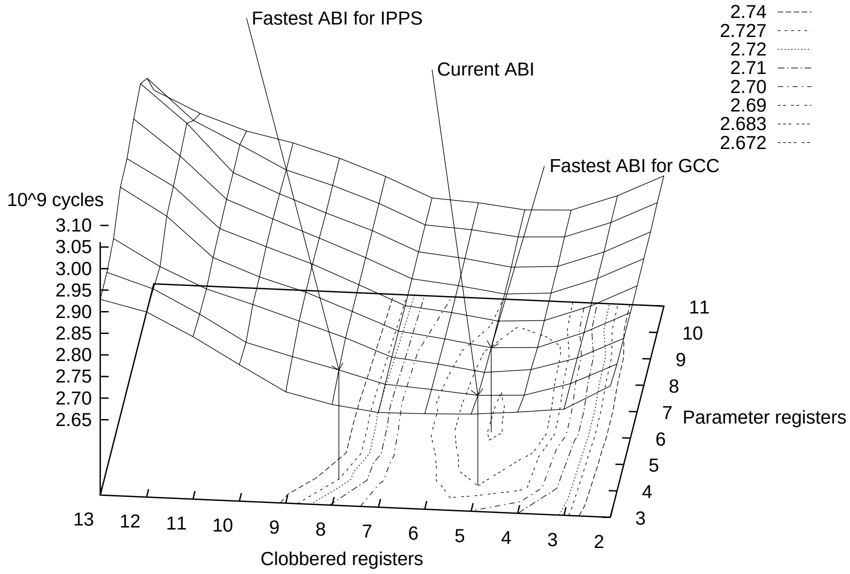

Figure 5.3: $t_c(p, c, \mathrm{GCC}_{\mathrm{combine.c}})$, time with *clobbered* parameter registers

changing the number of parameter registers judging from these measurements.

There is a "valley" in the execution time for both GCC and IPPS, corresponding to their optimal number of call-clobbered registers. For IPPS the valley has a distinct low at *eight* call-clobbered registers. GCC has a lower decline, where the fastest code corresponds to *five* call-clobbered registers, which incidentally is the same as for the current ABI.

5.8 Summary

The optimal values for IPPS and GCC touch upon those of the current ABI: GCC would optimally have *six* parameter registers and IPPS runs fastest with *eight* call-clobbered registers. The differences in execution between the optimal values for the current ABI and the one optimal for GCC are pleasingly small: the current ABI is only 0.45% slower, but for IPPS, the current ABI is 3.4% slower.

For IPPS, adjusting the number of call-clobbered registers may give a win, one could say it would be larger than the induced loss for GCC: a 1.1% in speed could be gained from increasing from *five* to *seven* registers. The loss for GCC would be 0.61% compared to the current ABI. This would even out the

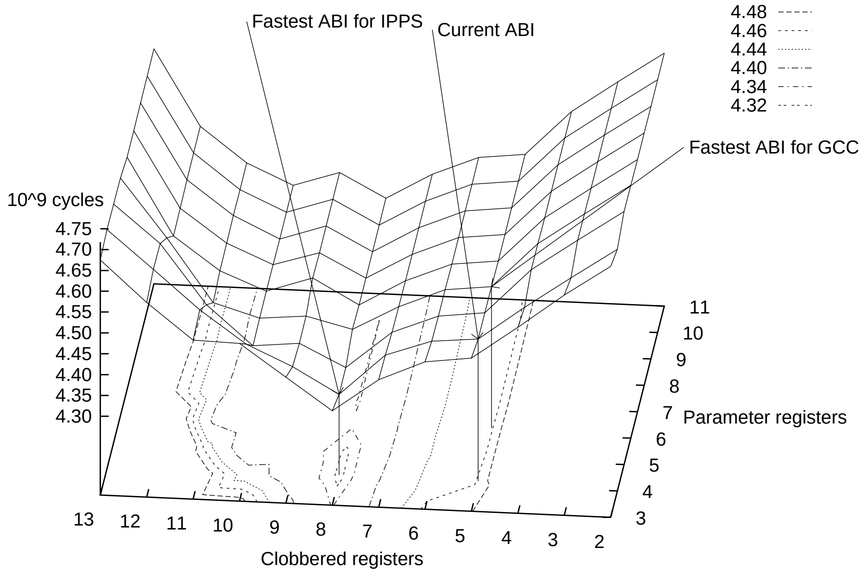

Figure 5.4: $t_c(p, c, \mathrm{IPPS}_{\mathrm{typical}})$, time with *clobbered* parameter registers

deviations from the optimal ABI:s somewhat to 1.05% for GCC and 2.3% for IPPS. The optimal *eight* call-clobbered registers for IPPS would win 3.3%, but would lose 1.7% in speed for GCC, giving a deviation of 2.1% from its optimal.

Still, as shown, there is no certain win for other applications from changing the ABI based on these observations — remember, the current ABI *is* within the application-specific margins.[12] I do not think these results should lead to an ABI change.

Which one of these programs comes closest to the typical code of a specific application is hard to see. The distribution of execution time in the code of these two programs would have been interesting to compare, as a tight "inner loop" would show up as making that specific piece of code important. The execution time would then differ significantly for different input and even small modifications to the program. On the other hand, this is true for most applications, so there's nothing new; it's just another program-specific behavior.

[12]As mentioned earlier, similar measurements for *C++* programs, would be interesting, and could very well point in other directions than these measurements.

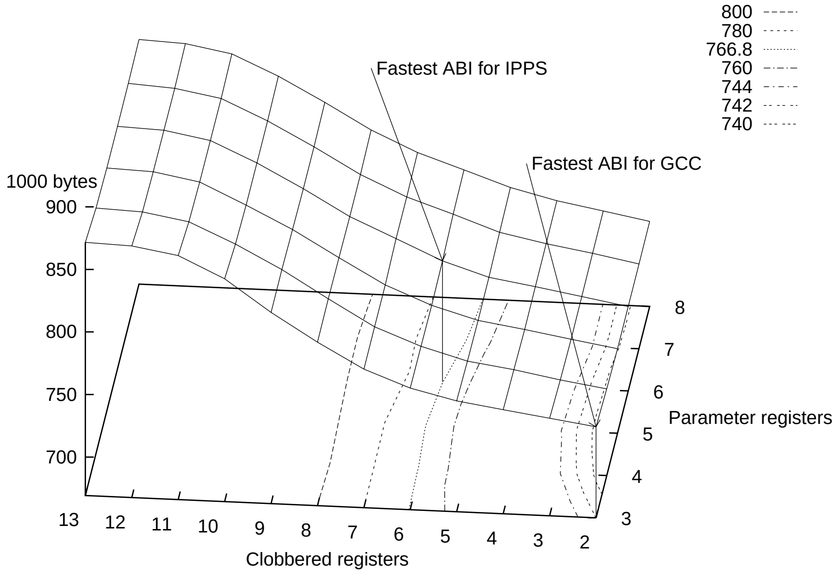

Figure 5.5: $s_s(p, c, \mathrm{GCC})$, size with *saved* parameter registers

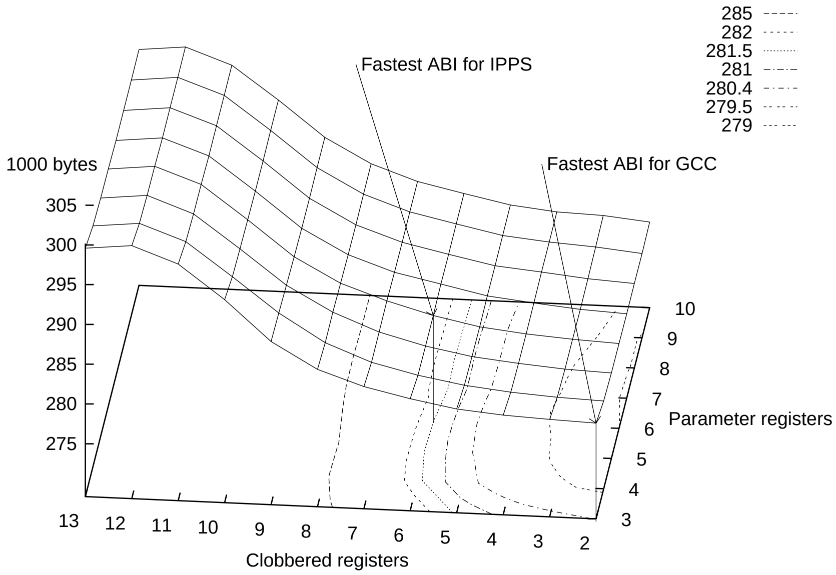

Figure 5.6: $s_s(p, c, \mathrm{IPPS})$, size with *saved* parameter registers

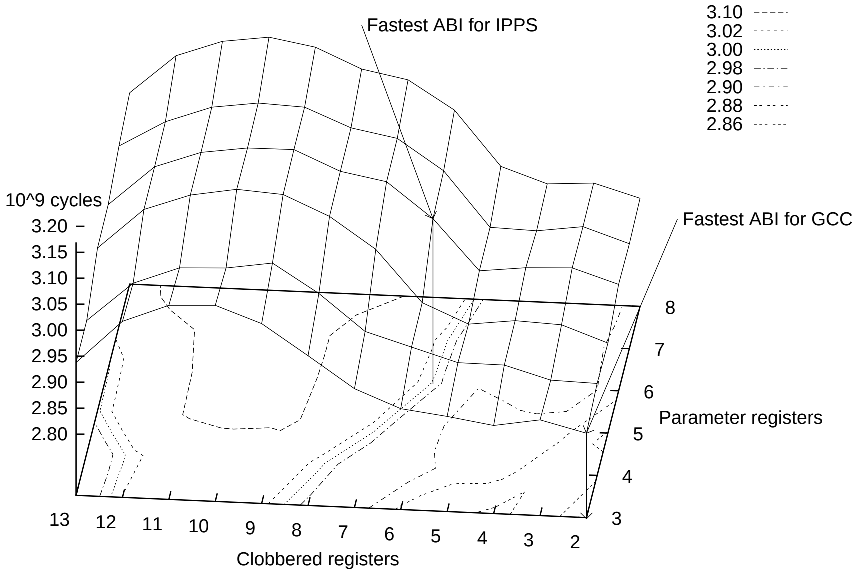

Figure 5.7: $t_s(p, c, \mathrm{GCC}_{\mathrm{combine.c}})$, time with *saved* parameter registers

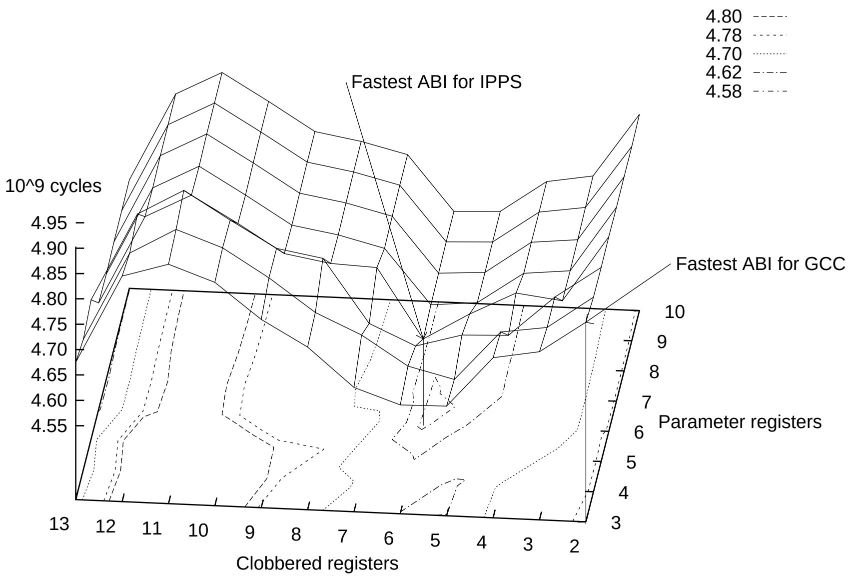

Figure 5.8: $t_s(p, c, \mathrm{IPPS}_{\mathrm{typical}})$, time with *saved* parameter registers

Chapter 6

Conclusions

GCC is sufficiently large that you can make it the task of a lifetime to improve it and to keep track of the related mailing lists and snapshots. Some parts need improvement more than others; the documentation is not the least. It is well worth attention for a career or maybe just as a hobby. It will not be "finished" in the foreseeable future: as new architectures emerge, GCC needs to be updated, and a lot of possible optimizations are currently just wishes.[1]

During this project, I made a lot of mistakes, came to a few conclusions on how porting *should* be done and reached some insight in how to write more portable and efficient *C*.

6.1 Some mistakes I made

Since you learn from your mistakes, the more the better:

6.1.1 Too smart

Sometimes I would find some of the generated code to be sub-optimal, something that looked like GCC needed fixing, so I fixed it in the port. For example, it seemed like GCC overlooked some obvious candidates for common-subexpression-elimination: the fact that a nearby object could be reached by a small offset from a symbol already located in a register was not used – the whole symbol was used in the address for the memory address. I therefore added code in `cris.c` to keep track of the register contents for the current basic block during assembler-output for those cases.

Well, it was a small win for that version of GCC, but improvements in the wrong place, such as this, gradually change into maintenance problems. It would have been better to just write the sub-optimality observation down, together with a test-case, or to fix GCC itself (although that option was not feasible at the time).

[1] Wishes of *contributions* from whomever has the incentive to code them.

6.1.2 Cramming the peep-holes

Another thing to avoid is tinkering with peephole optimizations. The situation appears like this:

Looking at compiled code, you see what you think are blatant assembly code optimization misses on GCC's part, optimizations that should be obvious to the compiler from your machine description. In the typical case, you ignore that they are normally rare enough to make a low impact on total performance. You want to fix them, so you do, using peephole optimizations. Then you find more, and the list of peephole optimization patterns grows longer. When it's time to check and update the port for that new GCC version (assuming you only do this from time to time), you have no idea why you put in most of those peephole optimizations; either because you don't have a test-case where it happens, or because a later improvement in GCC made the peephole-optimization superfluous, or both. In either case, you created a maintenance problem. My advice is to ignore the peephole optimizations until you have a rock-steady implementation of the compiler. Then you can start getting picky about the code.

But before adding any peephole optimizations, consider spending your time adding splitting-directives, combination patterns and defining and refining the `RTX_COSTS()`, `ADDRESS_COSTS()`, `CONST_COSTS()` macros. Do not forget operand discrimination in the `md` file, using the best instruction and addressing-mode e.g. for constants of different known values. However, remember to try and keep the port terse; avoid clutter.

And of course, for every peephole optimization or for *any* change, add a matching test-case to your regression test (see chapter 4.6), so you have the option to remove the peephole-optimization, once the compiler handles the optimization without specific guidance.

6.1.3 Bloating the macros

All macros in the `tm.h` file that are *C* expressions and not just trivial constants, should preferably be implemented as function calls, to functions in `tm.c` (at least when developing the port). If I had done that with *all* of them straight from the start, then recompilation and debugging would have been much easier. As it was, and still is in general, you have to either enable dependencies in the make-file for `tm.h`, causing a re-compilation of just about the whole compiler, or you do the work manually by removing just those object files that would be affected by your changes.[2] This is too error-prone!

6.1.4 Not setting the priorities right

I spent way too much time describing the 64-bit-entities, considering that there were major bugs in the general support for them in GCC 2.1. There was no real need for them at the time. I should have moved on to other actions as soon as I

[2]This is now less important, but just a few years ago, (the lack of) machine power made the compilation time of GCC a major factor in the development phase.

found out that there was generic problems that would show up in to the CRIS port as well.

6.1.5 Unpredictable predicates

CRIS has sign- and zero-extend capabilities on one operand for some of the standard arithmetic instructions.

At one early moment I tried to put the "extended" operand *including* the extend operator in a single operand to be matched by `match_operand`. The goal was to include it in the standard names for those arithmetic instructions. Unfortunately, this stymies the register-allocation-pass, that had to replace the entire operand with a register or simple memory operand. Such a simple operand always has to be allowed in the constraints for a `match_operand`, for that or another matching pattern.

It turned out that a much better description was to add anonymous names for those instructions, with the sign- and zero-extension explicitly stated, and the operands inside being matched normally.

6.2 How to port

If I get the chance to write other new ports, this is basically how I would do it, and how I believe an unexperienced porter would get the best result.

6.2.1 ABI

First, consider the ABI. If there is none defined and fixed in some way through previous work for that architecture, make one up. If you believe that an existing ABI is not optimal and you have the option to change it, keep it anyway and write down your suspicions for later.

Determine the fundamentals of your machine, like basic types and memory layout. Then, think up a way to call functions and how to return values. If you do not have a clear understanding of this, consider studying some standard ways, see [ABI:s]. Be prepared to revise your decisions when you have a working system, where you can perform measurements on running code. If you've never made a GCC-port before, chances are even greater that you will — unaware of it — make assumptions about optimality that can be proven wrong, so don't forget to check.

6.2.2 The machine description

Read [Stal 92] cover to cover. When looking through [Stal 92], write an over-simplified `md` machine description file, where you define only the most basic instruction patterns. Keep the patterns as clean as possible. Avoid special cases and instead take the penalty for a non-precise description. This is good for training, but also to realize what basic support your architecture needs in terms of descriptive C macros,[3] which you have to write before your first compilation.

If you have trouble figuring out how to specify something, look in existing ports, but try to avoid the level of detail that in many cases is there. It might be useful to start with one of them as a template for `tm.h` and `tm.c`, but the `md` file is best written from scratch, though.

6.2.3 Crossroads: decision details when porting

Where to put things that don't fit

Sometimes, there is a need to describe restrictions or features to GCC that do not really fit into the existing machine description framework.

The best possible solution is of course to extend GCC to include a better description of that feature, but more often than not, it is not clear what to do, and you probably want to try different approaches. Then by all means, if possible, start with modifying just the local port-specific files.

When this happens, do not try and abuse GCC mechanisms that were intended for describing other features. Even though it seems to work for the current version of GCC, this is probably just a coincidence, and it will probably change.

Anyway, especially when experimenting and *all other things being equal*, go for the following scheme:

- Put things likely to change in the `tm.c` file. That file is just recompiled, and re-linked together with the rest of the compiler, so you don't have to worry about making sure that the rest of the compiler get re-compiled.
- Put things in `md`. The compilation process checks this file and all parts generated from it thoroughly, so if you just change a comment, your compiler will not be completely re-compiled.
- The `tm.h` should be used only as a last resort.

  If you have to modify the rest of the compiler, `#define` guiding macros here, and conditionalize with `#ifdef`:s, to make the compiler execute *exactly* the same as before for other architectures.

PUSH_ROUNDING and friends

Where and how to pass stack-located parameters to functions, is a major issue. It may feel safe to always push and pop the parameters around function calls, but beware. If you make up your own ABI and any *push*-instruction is not faster than a *store*-instruction, it is recommended to go with the third alternative mentioned on page 34. This because you will save at least one `pop`-instruction for every call or chain of calls. The allocation and deallocation of stack at the function prologue and epilogue can probably (depending on the construction of

[3] See chapter 3.2.1

the stack frame) be merged with the allocation and deallocation of room for local variables.

6.2.4 Grease the port

After compiling a couple of test programs, the port should be checked against real programs, not only the c-torture test-suite.[4] Any uncertain details of the ABI and machine description parameters should be measured and checked to make sure that the port is sufficiently optimal, as described in chapter 4.6 and chapter 5.

6.2.5 Port portability

I believe there is no question that the most natural way to get a new GCC port up and running is by using GCC when you compile it. However, it is important to check every now and then that the port is still portable enough to be compilable with other compilers. This also makes sure that no GCC-specific features creep in, and that possible bugs in the port are exposed to two or more compilers, making it more solid.

6.3 Other ports

If you port a new architecture, you will probably study ports for similar architectures, or ports to architectures that you are familiar with. Most likely, this is one of the ancient ports, such as SPARC, Vax, i386 or MC68K.

Do not despair if you lose track; these ports may seem "cluttered" with details that are obfuscating when writing a "clean" new port. In some cases, it's because they have to cope with peculiarities of some assemblers, different assembler formats and different architecture variants. Other times, they may need a rewrite because of bit-rot. Things change in GCC; what generated the best code for the last version, may have been a short-sighted optimization that in the next version leads to a maintenance snake-pit.

Use the mature ports as guides, but make your own decisions.

6.4 How to write *C* for gcc-cris

Here are a few tips on how to write your code for portability but still tickle GCC to make the best code. Actually there are no really CRIS-specific tricks to teach, and these tips should also improve code for any compiler — not only GCC. At least, these tips shouldn't lead to *worse* code on other architectures or compilers.

[4] See appendix A.

6.4.1 Local variables

In a function, whenever there's a global variable or index into an array, or some scalar data that is accessed via a pointer and used more than once, *use a local, temporary variable*. Applied with care and descriptive names, this will make the code more readable as well.

This custom lets the compiler know that there's nowhere else where this data is accessed or changed, so it can safely assume that this data is constant over function calls or assignments. This improves code partly because it avoids deficiencies in the *alias* analysis of GCC (optimizations possible when different data does not overlap), and partly because of *real* "disambiguation" when data might overlap through different pointers or arrays. You might think that the compiler will run out of registers or use a lot of stack space for the local variables, but it doesn't, as long as the data is scalar and they are declared as needed, as described above. Actually, GCC does almost the same by itself before the register allocation pass, whenever the operands of an instruction do not match at the first attempt.[5]

Remember that the *C* standard does not allow the compiler to optimize access to any globally accessible data, if there is any chance that an assignment or function call (not including interrupt-functions) can modify it from one use to another.

6.4.2 Structures for global data

Local (and global) data that are defined next to each other (but not in a `struct`) is *not* automatically optimized for that locality. This means that even though two variables could be reached with a simple addressing-mode using a pointer and small index, no such optimization is attempted.

Therefore, if you group your local (`static`) non-function data in a `struct`, the code is improved. As a positive side-effect, a later modification to a plurality of the grouped data is made easier — often a major rewrite is needed to make a program work with *many* items when it was originally written to handle *one* item; be it serial ports or databases or computer screens.

6.4.3 Looping and pointers

There are deficiencies in the common-subexpression-elimination optimizations performed by GCC, which shows up for pointer arithmetic. This can be seen as an extreme case of the "always use local variables" as described above, but to the untrained eye it is not obvious where and how this happens.

Avoid using subtractive pointer arithmetic, and using the difference in number of elements. Such operations result in integer division of address-difference by element-size, which is more expensive than just a pointer comparison. This is bad for performance, especially when used as the end-conditional in a loop mainly handling a pointer. The following example code makes it more clear:

[5] See page 54.

```
struct foo { int hash; int num_elems; char *keys[41]; };

void
bar (int i, struct foo *fpb)
{
  struct foo *fp;

  /* Initialize i elements at fpb */
  for (fp = fpb; fp - fpb < i; fp++)
    fp->hash = 0;
}
```

This looks quite unsuspicious, but will in fact compile into a integer division being performed each round.[6] The following, almost identically-looking code will compile into much more optimal code:

```
struct foo { int hash; int num_elems; char *keys[42]; };

void
bar (int i, struct foo *fpb)
{
  struct foo *fp;

  /* Initialize i elements at fpb */
  for (fp = fpb; fp < fpb + i; fp++)
    fp->hash = 0;
}
```

Here, the `fpb + i` part is computed outside of the loop.

6.4.4 Inlining functions

GCC is able to by itself locate and inline such functions that would profitably be in-lined into other functions in the same file, when compiling with an optimization level of three and higher, or `-finline-functions`. They just have to be defined before they can be in-lined. If the code would not suffer in readability, try to order your code with smaller, often-used functions before the larger functions in the same file.

6.4.5 Dead strings

Watch out for unused constants and strings. GCC emits them even if the code that was supposed to use them is optimized away. For example, a common debug construct is:

[6] Which is actually optimized into an integer multiplication, but none the less; it is less optimized than the "additive" version.

```
if (DEBUG)
  fprintf (stderr, "Hello, bug!");
```

where the macro `DEBUG` is `0` for when no debug output is wanted. Of course, the test-of-zero and the call to `fprintf` is optimized out; but the string `"Hello, bug!"` is still left, taking up room in the code.[7]

Anyway, there's a kludge in the CRIS port to avoid unreferenced strings, so this tip does not really apply for gcc-cris. To implement dead-constant-removal at a generic level in the compiler core is feasible, but at the time it was not as easy as just intercepting string output, string-specific label definitions, and the use of those labels.

6.5 Limitations

The port and measurements have some limitations in the following areas, besides obvious cleanups for experimental code, unused code and formatting:

- Performance figures from *a run-time system as a whole* with interrupt- and process-handling capabilities are missing. However, if the system is well-written, it should have no great impact on performance compared to just running a single "batch" application, specifically no impact related to the compiler.
- Some kludges — or just impure solutions — are in the port. For example:
  - The patterns "overcoming" the `r15` $\equiv$ `pc` of the `casesi` expansion.
  - A problem with the eliminating frame-pointer with stack-pointer plus offset — the offset is based of whether the function is a leaf-function and calls functions with more than four parameters, and may have an extra "adjustment" instruction in the function prologue.
  - Some assumptions that a *float* on the host is in the 32-bit IEEE-754 floating point format.
- No test-cases for those bug-fixes (or rather workarounds) that make some macros and machine descriptions look strange.
- The `..._COSTS` macros could be optimized (this would necessarily include measurements).
- The decisions on the ABI was based on somewhat vague assumptions from observations in static code, not real measurements on running code. Although this is a good technique for starters, the ABI design should have been revised based on observations such as those presented in chapter 5.[8]

[7] This may be improved soon enough, but it is still true for gcc 2.8.1 and egcs 1.1

[8] The difference in use of the ABI in the currently dominating language *C++* makes the measurements less important; instead comparative complete *C++*-programs should be added and measured besides IPPS and GCC. Anyway, this report deals only with the *C*-side of the compiler story.

But that is history now, and the differences from the results of the measurements in chapter 5 aren't distinct enough, let alone big enough to warrant a change.

- The target macros related to exceptions as used in *C++*, are not defined as should be done in the CRIS port. However, exceptions were just not usable in version 2.7.2, and are not used in *C* code anyway.

6.6 Summary

- The port works, and is a major contributor to the success of the CRIS architecture and the ETRAX chip series. For *C* code, there are no major restrictions or left-out features in the port that are used in an embedded system.[9]

- Porting GCC was in practice a lot harder than theory would make you believe — it is not enough just to write a good machine description.

  To a large extent, the ability to debug the GCC core and reading and understanding (well, at least following) the source code of GCC, makes a vital difference. It means the difference between the situation where a blindly patched port works for simple test-programs but crashes for any sufficiently complicated function, or where a debugged and tested port works for all encountered code.

- The compiler is not enough. You have to make sure, or plan for, a simulator or real hardware with debug functionality, an assembler, linker and run-time library. If not, you will use lots of unexpected extra time to make those work. There is no short-cut; small self-contained programs are not enough to sweat out the bugs in a port.

[9] For example, *position-independent code* is quite useless when there is a single address space and you know where the code ends up.

Appendix A

How to get the code for different programs mentioned herein

Most programs mentioned in this report are GNU programs; developed as free open source under the GNU GPL licensing (see <URL:http://www.gnu.org/philosophy/free-sw.html>).

These programs and others:

- GNU make
- GCC (and its different language front ends)
- emacs
- ghostscript (a postscript interpreter)
- perl (a programming language)
- diffutils (diff generates easily viewable differences between text files)
- patch (applies differences in the *diff* format)
- glibc (a runtime library)
- binutils (assembler, linker and such)

are available from sites listed at <URL:http://www.gnu.org/order/ftp.html> or <URL:ftp://ftp.gnu.org/pub/gnu/GNUinfo/FTP>.

Another runtime library for embedded systems, newlib, can be reached at <URL:ftp://ftp.cygnus.com/pub/newlib/>.

The GCC/egcs regression test suite is available as part of the egcs packages, see <URL:http://egcs.cygnus.com> and

<URL:ftp://egcs.cygnus.com/pub/egcs/releases/index.html>, or as an older, smaller, but more free-standing version at
<URL:ftp://vger.rutgers.edu/pub/gcc/c-torture-1.45.tar.gz>.

Most programs are stored in the compressed *gzip* format of a file bundle in the *tar* format, generated by the programs with those the same names. If you are not accustomed to those formats, contact your local friendly Unix-person, or use your favorite web search engine to find out how to use the files on *your* platform.

Appendix B

The gcc-cris code

... is not included in this report. I do not see any reason to include a paper copy of the port, considering the size of the code (c:a 16000 lines), and the widespread availability of Internet connections.

However, the code is electronically available over Internet. The latest version of the compiler is located at
<URL:ftp://ftp.axis.se/pub/axis/tools/cris/misc/gcc-cris.tar.gz>.

Different versions will be located at the same directory, with the naming convention `gcc-cris-`*gcc-version*`-R`*target-version*`.tar.gz`, as in
`gcc-cris-2.7.2-R3.tar.gz`. Patches consisting of the CRIS-specific differences (using `diff -c3prN`) from the original GCC distributions are also here; replace "tar" with "patch" in the file name. This will save you some time when downloading, if you already have a copy of the original distribution, or just want to take a look at the CRIS port.

Note that these distributions only consist of the GCC compiler with the CRIS code added and with some necessary modifications to the core compiler (very few, and documented in the `ChangeLog` file in the distribution). If you want to get hold of a complete compiler system for CRIS, look in
<URL:ftp://ftp.axis.se/pub/axis/tools/cris/compiler-kit/>.

And yes, my intent is to make the CRIS ports of the compilation tools, specifically GCC, be a part of the official distributions.

Appendix C

GCC, LCC and other compiler information

There is quite a lot going on about free compilers on the net.

C.1 GCC

C.1.1 Mailing lists

gcc2 To subscribe to the GCC developers list, send an email with an empty subject, and only a single line saying `subscribe gcc2` *your-email-address* to `gcc2-request@cygnus.com`.

crossgcc To subscribe to the GCC- and embedded systems-related cross-compiler list, do as above, but substitute `crossgcc` for `gcc2`.

egcs These above are the "classic" gcc development lists. A more intense experimental GCC project, *egcs*, was formed in 1997 and has gained large momentum in development. See <URL:http://egcs.cygnus.com/lists.html> for the related mailing lists.

C.1.2 Newsgroups

A few newsgroups handle GCC-specific topics.

gnu.gcc.announce Mostly announcements about new versions.

gnu.gcc.bugs Bug reports to `bug-gcc@gnu.org` will end up here.

gnu.gcc.help Posts asking for help with GCC-related stuff.

C.1.3 WWW

- Read all about egcs at <URL:http://egcs.cygnus.com>.
- There is an official GCC page at
  <URL:http://www.gnu.org/software/gcc/gcc.html>,
  but there's no real news there.
- The GNU/FSF bug-tracking-list (which includes GCC) is located at
  <URL:http://www-gnats.gnu.org:8080/cgi-bin/wwwgnats.pl>.
- For locations of GCC sources, see
  <URL:http://www.gnu.org/order/ftp.html>
  or <URL:ftp://ftp.gnu.org/pub/gnu/GNUinfo/FTP>.

C.2 LCC

C.2.1 Mailing lists

Send a message saying `subscribe lcc` in the body (the subject is ignored) to `majordomo@cs.princeton.edu`. For the bug-related mailing list, substitute `lcc-bugs` for `lcc`.

C.2.2 WWW

The LCC homepage is at <URL:http://www.cs.princeton.edu/software/lcc>. Read <URL:ftp://ftp.cs.princeton.edu/pub/lcc/README> for up-to-date information on the source code distribution.

C.3 Other compilers

A thorough up-to-date list of free compiler-related resources can be found at: <URL:http://www.idiom.com/free-compilers/>.

Appendix D

General concepts, notation and terminology

The terminology used in this report matches or at least does not contradict with that of [Stal 92].

- A *byte* is an (unsigned) octet of eight bits (as usual). The explicitly signed equivalence is an *sbyte*.
- A *word* is a 16-bit entity; two bytes. In a context where twos-complement signedness makes sense, it is assumed signed. The unsigned equivalence is an *uword*.
- A *double word*, or *dword*, is a 32-bit (assumed signed) entity. The unsigned variant, where that matters, is an *udword*.
- General registers in CRIS are denoted `r0` to `r15`.
- *Byte-addressable* means that the address counts individual bytes; not necessarily that each byte is individually accessible.
- The *mode* of some data is the size together with the type. Often used in [Stal 92].
- An *architecture* is the type of the processor system; the set of processor-related resources of a system, and the way they are connected. Often abstracted to higher orders: a MC68K-type-architecture is a register-oriented architecture is a von Neumann architecture.
- The words *machine*, *processor* or *CPU* all denote the central processing unit (within a von Neumann architecture); where instructions and data are processed by a central unit in a clocked manner.
- The notation of all numbers is decimal unless otherwise told. A prefix of *0x* denotes hexadecimal notation, as in *0xff* for the decimal number 255.

- *Endianness* is the order in which smaller storage components are ordered within larger storage components. For example, *big-endianness* for bytes means that bytes are ordered within words with the most significant byte at the same address as the address of the word, and the least significant byte at the next higher address. Consequently, *little-endian* or *small-endian* for bytes means the reverse. Example: the word 256 (0x100) is stored byte-wise as $\{1, 0\}$ for a byte-big-endian machine, and as $\{0, 1\}$ for a byte-little-endian machine. Normally, a machine has the same endianness over most information entities, and are therefore called little-endian or big-endian architectures.

  Since most architectures are byte-addressable, the endianness normally does not relate to bits. However, the numbering of bits within a byte denotes the bit-endianness. Most machines address bit 0 within a byte as the least significant bit, and are therefore bit-little-endian machines, regardless of the general endianness.

- A *port* is the result of work involved in extending a program to be runnable or usable for a new system. The work itself is known as *porting*. When a normal application is ported to a new system, this means that it can run on the new system.

  Ports of compiler-type program may need to be more specific. A *host* port is when the program is ported to *run* on another system), which may be different from a *target* port, where a program is ported to *generate code* that runs on another system. See cross-compiler.

- A *cross-compiler* compiles code for another type of system than the one where the compiler runs.

- An *application binary interface* or *ABI* for short, is the set of calling conventions and the memory layout of basic data types; for example how parameters are passed to a function. You have to invent this yourself for a completely new system, but most often it is fixed, due to previous work on that system.

- The *front end* of a compiler parses the compiled language (lexical and syntactical parsing), and the resulting syntax tree is passed on to the compiler back end.

- The *back end* of a compiler takes an intermediate representation of the program, may it be a syntax tree or a three-address representation, then generates the compiler output, at assembler-code-level. The general limits of front and back end are vague. In GCC, the front end takes care of the language parsing, and the back end handles topics related to the actual architecture where the compiled code will run. Sometimes, the back end is referred to as meaning the target-specific files, other times just generally everything but the front end.

- The terms *function* and *subroutine* will be used interchangeably, meaning a common piece of code in a program, that can be used (called) from various positions in the program without duplicating source code, or for that part, object code. (Hopefully the reader is already familiar with the concept of functions and will not be confused by this explanation!)
- The basic types and any type of pointers are *scalar data*.
- Data supplied specifically in the call to a function/subroutine, are called *arguments* or *parameters*.
- A function that does *not* call any other functions, neither explicitly as visibly in the code, nor implicitly as a library call for a mathematical or standard builtin operator, is called a *leaf-function*.
- A *boundary* for anything addressable in memory, means that natural division of addresses. For example, a 16-bit boundary means an even (byte-addressed) address, and a 32-bit boundary means an address that is a multiple of four.

Bibliography

[Stal 92] Richard M Stallman, Using and Porting Gnu CC, (Supplied in electronic format with the source code, or see appendix B). Free Software Foundation 1998. ISBN 1-882114-37-X

[Axis 95] Per Zander. CRIS Instruction Set. Axis internal document (later version available in printed form as “CRIS Programmers Reference”).

[make] Various. Do `man make` to find out. Should be on your Unix-flavor system. The “info”-pages of GNU make are available through the WWW at various sites, for example at my institution:
<URL:http://www.efd.lth.se/cgi-bin/info2www?(make.info)>

[RedDragon] Aho, Sethi, Ullman. Compilers: Principles, Techniques and Tools. Addison-Wesley 1986. ISBN 0-201-10194-7

[K& R C] Kernighan, Richie. The C Programming Language, Second Edition. Prentice-Hall 1988. ISBN 0-13-110362-8

[ABI:s] Good overviews are hard to find. Here’s some of the best I found through <URL:http://altavista.digital.com> (Please omit the “-” at line-breaks in the URL:s):

- There is one specified for MIPS processors:
  <URL:http://www.mipsabi.org/Tech/BB3.0/bbtoc.htm>.
  See specifically chapter 3.
- A graphic comparison between calling conventions for MC68K and PowerPC for Apple’s MacIntosh:
  <URL:http://www.mech.uwa.edu.au/HowTo/intro_to_ppc/-ppc4_runtime2.html>.
- Overview of the Watcom compiler, ABI chapter:
  <URL:http://lili.iit.uni-miskolc.hu/stuff/doc/help/-product/watcom/compiler-tools/ccall32.html>
- Description of the Introl C compiler:
  <URL:http://www.introl.com/introl-demo/Reference/-compiling.html>.

[LCC] Christopher W. Fraser, David R. Hanson. A retargetable Compiler: Design and Implementation. Addison-Wesley 1995. ISBN 0-8053-1670-1