Technische Universität Wien

Diplomarbeit

Creating a GCC Back End for a VLIW-Architecture

Ausgeführt am Institut für Computersprachen
der Technischen Universität Wien

unter Anleitung von Ao.Univ.Prof. Dipl-Ing. Dr. Andreas Krall

durch

Adrian Prantl

Neustiftgasse 45/12
1070 Wien

7. Mai 2006

Unterschrift

Abstract

The Control Processor is a 24-bit, 4-way very long instruction word (VLIW) processor, developed by On Demand Microelectronics. In this work, a port of the back end of the GNU Compiler Collection (GCC) is introduced that takes full advantage of the CPU's parallelism and generates parallel assembler code. Also, the GNU Binutils and the Newlib C runtime library were adopted to support the Control Processor.

The presented GCC back end uses a simple pipeline description to model the functional units of the Control Processor for the instruction scheduler. Based on the results of the scheduler, a separate pass assigns the instructions to the slots of a VLIW bundle. Using this technique an average utilisation of up to 2.5 instructions per bundle is achieved. Special care was taken to support the Control Processor's unusual byte-lengh of 24 bits, which affected many design decisions.

In a second part, the existing assembler for the Control Processor was extended to create object files in the Executable and Linkable Format (ELF). To create a linker and other utilities for these object files, the GNU Binutils were ported to the new target. Based upon this, the GNU Debugger was also extended with an interface to the instruction-level simulator of the Control Processor.

Finally, the C runtime library Newlib was ported to the new target as well, thus completing the cross-development environment for the Control Processor.

Kurzfassung

Der Control Processor ist ein 24-Bit, 4-fach VLIW (very long instruction word)-Prozessor, der von On Demand Microelectronics entwickelt wurde. In dieser Arbeit wird eine Portierung der GNU Compiler Collection vorgestellt, die in der Lage ist, den Parallelismus des Prozessors voll auszunützen und parallele Assemblerbefehle auszugeben. Weiters wurden die GNU Binutils und die C-Laufzeitbibliothek Newlib an den Control Processor angepasst.

Das vorgestellte GCC-Backend verwendet eine einfache Pipelinebeschreibung, um die Funktionseinheiten des Control Processors für den Instruction-Scheduler zu modellieren. Anhand der Resultate des Schedulers werden die Befehle in einem eigenen Durchlauf auf die Slots eines VLIW-Bündels aufgeteilt. Durch diese Methode wird eine Slotausnutzung von bis zu 2,5 Befehlen pro VLIW-Bündel erreicht. Eine besondere Herausforderung stellte die ungewöhnliche Wortbreite des Control Processors von 24 Bit dar, die das Design des Backends entscheidend geprägt hat.

In einem zweiten Teil wurde der bereits vorhandene Assembler des Control Processor um die Fähigkeit erweitert, ELF-Objektdateien (Executable and Linkable Format) zu erstellen. Weiters wurden die GNU Binutils auf diese neue Plattform portiert, um den darin enthaltenen Linker mit diesem Format nutzen zu können. Darauf basierend wurde auch der GNU Debugger um die Möglichkeit erweitert, den Simulator für den Control Processor direkt einzubinden.

Um die Entwicklungsumgebung zu vervollständigen, wurde auch die C-Laufzeitbibliothek Newlib für den Control Processor angepasst.

Danksagung

Ich möchte mich an dieser Stelle vor allem bei meinem Betreuer Professor Andreas Krall bedanken, der mich während des Designs und der Implementierung des GCC-back ends aber auch später bei der Erstellung dieser Diplomarbeit immer unterstützt hat.

Mein besonderer Dank gilt auch dem gesamten Team von On Demand Microelectronics, allen voran Karl Neumann, der mich während unserer Zusammenarbeit stets ermutigt und viele hilfreiche Ideen beigesteuert hat. Außerdem möchte ich mich bei Julia Ogris bedanken, die mir als Autorin des Simulators sehr viele Fragen beantworten musste.

Schlussendlich möchte ich mich bei meinen Eltern bedanken, die mich bei meinem Studium unterstützt und meine Neugierde gefördert haben, bei meiner Freundin und allen meinen Freunden und Studienkollegen mit denen ich die letzten Jahre sehr gerne verbracht habe.

Contents

List of Figures

1. Introduction

The Control Processor is a 4-way VLIW[1] architecture aimed at parsing next-generation video streams like H.264. The Control Processor was designed by the Vienna-based On Demand Microelectronics who is specialised in high profile digital signal processing applications.

A VLIW architecture is basically a statically scheduled version of a superscalar RISC[2] processor, where the burden of assigning the instructions to execution units lies on the programmer. This has the advantage of eliminating the dispatch logic in the processor and being able to spend more time finding an optimal distribution for the instructions.[3] Programming such a machine in assembler is more challenging compared to a classic RISC processor, because the programmer has to take extra care not to introduce dependencies inside an instruction word. For this reason, it is favourable to provide the programmer with a compiler that will take care of finding the optimal allocation of functional units with instructions and let the programmer specify the tasks of the processor in a high-level language such as C. The high-level language approach is also an advantage for the experimental evaluation of architectural parameters such as the number of parallel functional units in a CPU. A compiler can be constructed in a reasonably configurable way such that changes made to the hardware can be reflected in a relatively short timeframe.

The goal of the work presented in this document was to create a cross-development environment for the Control Processor and the C programming language. Already available was an assembler and an instruction-level simulator that could be incorporated in the final toolchain.

The GNU compiler collection was an obvious choice for the task, as it is available in source code, relatively well documented and designed to be highly extendable. Another important point is that it is well integrated with the rest of the GNU toolchain which includes the GNU assembler and linker, but also the widely used GNU debugger, a runtime system and many operating systems. The debugger is an important component of the overall development environment, because it contains the bridge to the instruction level simulator which in turn is necessary to perform any serious testing and development in the absence of actual silicon hardware.

A personal goal that I have for this work is that it serves as a documentation of the steps that are necessary to create a complete development environment for a new processing platform. Most of the components of the toolchain are well documented, but there was not too much documentation about their interaction and the dependencies

[1] very long instruction word
[2] reduced instruction set
[3] [PH98], chapter 6.12, pg. 528

they impose during the development. Wherever possible references to further existing documentation are included throughout the text, so it can be used as a guide during the creation of a new GCC back end.

In the first chapter, the architecture of the On Demand Control Processor is introduced. The description of the toolchain is structured into chapters that discuss each package in detail, at first with a general overview the tool's the mode of operation, then the hands-down description of steps necessary to adopt it to a new architecture, as well as common pitfalls that are to be avoided. Finally, the complete toolchain is presented and some examples of the quality of the final compiler are given.

2. The On Demand Control Processor

The On Demand Control Processor is part of On Demand's scalable video engine (`SVEN`). The Control Processor is meant to be used as a bitstream decoder and in its basic configuration it is powerful enough to handle the parsing of H.264, VC-1 and MPEG2 streams. These are popular formats for the digital distribution of TV broadcasts and feature films.

Its main features are a 4-way VLIW core, 64 general purpose registers, conditional execution, support for various extensions and a 24 bit address space.[1]

2.1. Architecture

The Control Processor is in essence a load/store architecture with a reduced instruction set (RISC) and in-order execution. Special to it are four mostly independent instruction units which are fed by very long instruction words (VLIW). These processing units are called slots. It is up to the assembler programmer and the compiler to properly distribute the single instructions across the slots.

In its standard configuration, the processor has a total of 64 general-purpose registers, each with a length of 24 bits. These parameters are meant to be configureable. The pipeline is transparent to the programmer and has three stages: Fetch, Decode and Execute/Writeback. Two of the four slots can be provided with a condition which is evaluated by one of the other two slots. The arithmetic units of the Control Processor do not support multiplication or division of integers; these features are not needed in its key field of application. Of course there is no native support for floating point operations either.

Data memory and program memory live in separate address spaces. The CPU can address 2^{24} independent words in each address space. While one data word is 24 bits long, one instruction word is significantly longer, since it has to hold 4 instructions, including all operands.

The Control Processor supports several extensions for specialised tasks that come up when decoding modern video streams. These include algorithms like Variable Length Coding (VLC), Content Adaptive Binary Arithmetic Coding (CABAC), Content Adaptive Variable Length Coding (CAVLC), and Exponential Golomb Coding which are necessary to decode H.264 streams for example. These extensions are accessed through the port interface, where they appear as data memory addresses and can be made available through library functions in a compiled language such as C.[2]

[1] see [Win05], pg. 7
[2] see [Win05], pg. 8ff

2.2. Instruction pipeline

The pipeline of the On Demand Control Processor has three stages which are not visible to the assembly programmer. There are no branch delay slots, but jumps force the pipeline to be emptied, inducing a penalty of two cycles.

- *Fetch:* The first stage is responsible for accessing the code memory in order to fetch the next instruction word.
- *Decode:* In the Decode stage data addresses for the more complex addressing modes are generated and the condition is evaluated. Conditions may be placed for slot 0 and 2 and are executed in slot 1 and 3.
- *Execute:* In the final stage the actual instructions are executed and a possible data memory access is performed. Conditional instructions are only executed if their condition evaluates to true.

The following restrictions emerge from the layout of the pipeline: Only one unconditional jump is possible per bundle. Two conditions may be applied and occupy an extra slot per condition, so up to two conditional instructions are can be placed into an instruction bundle.[3]

2.3. Instruction format

The Control Processor uses very long instruction words with 4 instructions per code word in its default configuration. These four instructions are also called an *instruction bundle*. One instruction consists of 6 fields: The operation code (OpCode), an address field, two source registers, a destination register and an immediate field. Both address and immediate value can be up to 24 bit long.

Slot 0						Slot 1	Slot 2	Slot 3
OpCode	Address	Src0	Src1	Dst	Immediate	...	...	...

Table 2.1.: Format of a very long instruction word

This layout can lead to several inconveniences when using standard tools. The values should be packed in a way that the total size of an instruction word is a power of 2. To perform relocations, the GNU linker needs to convert program-specific word addresses (and especially the immediate field within each micro-instruction) to 8-bit byte addresses inside an object file; if they cannot be converted through a combination of bitshifts and adds, a workaround has to be found. This problem does not exist in the actual hardware implementation; it simply uses a special uncached 216 bit memory for the instruction words.

[3] see [Win05], pg. 9ff

2.4. Addressing Modes

The On Demand Control Processor supports a total of four different addressing modes, which include absolute immediate addresses and register relative addresses.

- *Immediate:* The Address is given as an absolute constant integer.

  ```
  r0 = port[32];
  ```

- *Register:* The Address is the contents of a register.

  ```
  r0 = port[r62];
  ```

- *Register + Immediate:* The Address is the sum of a register and a constant integer.

  ```
  r0 = port[r62 + (-1)];
  ```

- *Register + Register:* The Address is the sum of two registers.

  ```
  r0 = port[r62 + r2];
  ```

These addressing modes apply to every command in the group of memory read/write instructions. Due to its load/store architecture, the On Demand Control Processor does not allow memory operands to any other instruction. The `jump` and jump subroutine (`jsr`) instructions allow only for constant operands.[4]

2.5. Assembler syntax

The syntax of the Control Processor's assembler is different from that of most general purpose CPUs, but it is very programmer friendly and easier to read than most common assembler languages.

The most obvious difference is that operations are not written in prefix but in an infix notation making it look more like a programming language than an assembler. For VLIW instructions a mechanism to mark the beginning and end of an instruction bundle is necessary; this is achieved by putting curly braces around every four operations. Conditions are expressed through the `if` keyword. Assembler source files are preprocessed by the standard C preprocessor (`cpp`).

Data declarations look like typeless C variable and array declarations. The assembler knows only one data type which is a 24-bit, two's complement, signed integer. It does allow the declaration of initialised arrays though.

Figure 2.1 shows a minimalistic example of the assembler's syntax. The fancy syntax has its downsides, too; the performance of the assembler is not as great as the typical GNU assembler implementation that has to parse a much simpler grammar. During the porting of the GNU toolchain, the different syntax was the reason why On Demand's existing lex/yacc based assembler was used instead of the GNU assembler.

[4] see [Neu05] and [Ogr05]

```
.data
     foo = 1;
     bar = { 42, 1, 0, 1, 2 }; // Array
.code
    {   // instruction bundle
        r0 = r1 + 1;       // slot 1 (add)
        ;                  // slot 2 (nop)
        if (r3 < 0)        // slot 3 (condition)
            port[foo] = 0; // slot 4 (memory access)
    }
```

Figure 2.1.: Example of the Control Processor's assembler syntax

3. The GNU Compiler Collection

3.1. Overview

Work on the GNU Compiler Collection (GCC) began in 1984 when Richard Stallman founded the Free Software Foundation and the GNU Project. The goal of the GNU Project has been to create a software environment consisting entirely of free software. Stallman started with the now famous Emacs text editor and a shell command interpreter, but it was quickly becoming clear that to write free software for a free system there had to be a free compiler. To fill this gap, the GCC project was started. While the GCC started as a C-compiler for the Motorola 68000 CPU it was always designed as a multi-language, multi-platform compiler. It was first released in 1987.[1]

From then the project has grown enormously and by the end of 2005 it supported over 60 host platforms, programming languages such as C, C++, Objective-C, Objective-C++, Java, Fortran and Ada and code generation for almost 40 different architectures.[2] These numbers only cover the official distribution. There are quite a few language front ends (such as PASCAL) and architecture back ends that are maintained separately.

3.1.1. Different configurations of the GCC

The GCC can be configured and installed in different ways, depending on the target platform and the way the compiler is to be used. The configuration system - based on the GNU Autotools - distinguishes three components that define a platform.

- Build System - This is the system on which GCC will be compiled on. The `configure`-script will autodetect this parameter.
- Host - This is the system on which GCC is going to run on. It can be specified with the `-host=` parameter. This parameter decides which compiler will be used to compile the GCC.
- Target - This is the architecture for which GCC will compile code. It can be specified with the `-target=` parameter.

The combination of build system, host and target system determine the type of compiler that is to be build. There are three configurations:

- *Native compiler:* This is the default configuration. In this case the compiler is intended to be run on the same machine as it is being built on and will also

[1] see [Sta06]
[2] see [Fou06b]

generate code for this very system. The default system compiler is configured in this way.

- *Cross compiler:* This is the common configuration for developing embedded systems, especially when the development machine is more powerful than the target machine. The compiler will generate code for a different architecture than the one it is running on. This configuration is also useful to build a portable project for many architectures at once.

- *Canadian cross compiler:* This more exotic case describes a cross compiler being built by a cross compiler.[3]

Host and target of the new compiler can be specified as flags to the `./configure` script of the GCC. As with all GNU development tools, GCC expects to be built in a different directory than the source lies in. This has many advantages: Aside from having an uncluttered source directory, it is also possible to create two different compilers from the same source without having to reconfigure the sources every time.

3.1.2. Important components of the GCC

The GNU Compiler Collection consists of many separate tools which work together during the compilation of a program's sources. The program the user will interact with most of the time is called `gcc` and is the compilation driver. It is a front end that collects all command line parameters and calls the respective tools in the correct order. Usually a compilation of a C source goes through the following stages:

- `gcc:` This is the user-visible front end that decides which programs to call. It is responsible for decoding the command line parameters and watches over the different stages of compilation.

- `cpp:` The C preprocessor. It expands C macros such as `#include` directives and is used for C, C++ and some assembler files.

- `cc1:` The actual C compiler. Its mode of operation will be discussed in more detail in the next section. Basically, it translates C source into assembler output.

- `as:` The assembler for the target machine. Its job is to parse the target's assembly language and to create a relocateable binary image or *object file*. The resulting object file will still contain references to symbols that should be defined elsewhere and a list of all addresses that have to be changed when the program will finally be loaded into memory.

- `collect2:` This tool scans objects for necessary initialisations at startup time and adds pointers to initialisation functions to a table so they can be called before the `main()` method is executed at runtime.

[3]The name "Canadian" derives from Canada having three major political parties in the 1980ies.

- `ld`: This is the GNU Linker and (like the assembler `as`) it is actually not a part of the GCC distribution but part of a package called `Binutils`. The GNU `Binutils` are a collection of utilities that handle manipulations of object files such as the the assembler, linker, the `strip` and `objcopy` tools.[4] The linker collects all necessary object files and libraries (which in turn are simply archives of multiple object files) and copies them together to form the final executable binary. It most importantly also resolves all references to symbols between all those object files.

Thankfully, not all of these tools have to be rewritten when adding a new machine. In order to support a new target, mostly the code generation back end of `cc1` and the assembler need to be extended.

3.1.3. Compilation with the GCC

In this section the process of compiling a (C) source file into assembler will be discussed in detail. A compiler can be divided into three main components: The language front end which understands the source language and constructs a parse tree, a middle layer working with an intermediate representation where optimisations and other program transformations take place and a target machine specific back end handling the actual generation of assembler code.

The compilation of a program happens in many different phases that are too many to describe in detail in this section. The majority of the passes during a compilation are introduced by the optimiser and have to be manually enabled through command line options. For the author of a back end the details of the optimisation passes are not too important, although it must be noted that some errors in the machine description only become apparent with optimisations enabled. The behaviour of the function stack frame allocation is an example for this, as the frame pointer may be omitted and local variables may or may not be stored on the stack, depending on the optimisation settings.

When debugging the machine description it can be helpful to have at least a basic understanding of what happens in the passes that are relevant for the back end. After most of the RTL-based transformations, the GCC provides a dump of the current state of the RTL representation of the program. This feature can be enabled with the `-dall` command line option. As these dumps are in RTL form[5], they only reflect changes made by the (RTL-based part of the) optimiser and by back-end-specific passes like the scheduler. A graphical overview of these passes, dataflow inside GCC and the intermediate representations is given in figure 3.1. The following passes are especially relevant to the author of a new back end.

expand: This is the first step of the RTL-based component. Here, the intermediate representation is expanded from GIMPLE trees created by the front end to the linear and more machine-near RTL expressions. The *expand*-dump therefore contains the first attempt at converting the source language into the intermediate representation used by the machine description.

[4] The `Binutils` should therefore be installed before trying to compile GCC.
[5] An overview of the intermediate representations in GCC is given in chapter 3.1.3

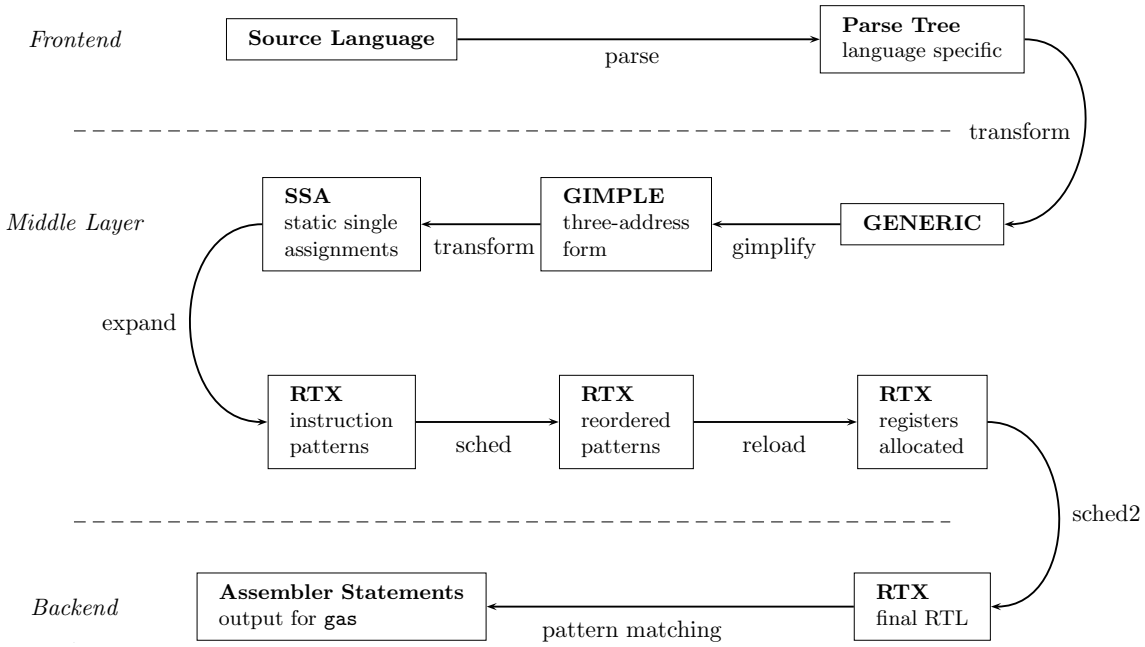

Figure 3.1.: Some passes and intermediate representations of the GCC

sched: The instruction scheduling pass. Here, instructions are reordered to reduce the number of pipeline stalls on processors with instruction latencies, as is the case with many RISC architectures.[6]

reload: The reload pass is where register allocation takes place. This pass marks a dividing line; after the reload is complete, instruction definitions may not introduce any more pseudo registers. The reload process spans a local (in the context of basic blocks) and a global phase.

sched2: Instruction scheduling is performed twice; before and after register allocation. The scheduler dump contains the final allocation of the CPU-units defined in the pipeline automata.

Dumps are also generated for all the optimisations such as common subexpression and dead code elimination. But for the back end above passes should be the most relevant.[7]

Intermediate representations within the GCC

The GNU Compiler Collection uses many different kinds of representations for program code. In the front end, the programming language is parsed by the compiler and the

[6] see [S+05], chapter 8.5
[7] see file gcc/passes.c in [Fou05]

program is stored in a syntax tree. This tree still carries language specific information which is gradually lost during the compilation.

Since the middle layer has to work with every supported language, a language independent representation is constructed by the front end. These trees are called GENERIC. They represent an entire function in an intermediate language.[8]

The GENERIC trees are then converted to so-called GIMPLE trees. GIMPLE is a simplified subset of GENERIC that is used during the optimisation passes.[9] That name is derived from the SIMPLE[10] trees of McGill University's McCAT compiler which they are based on. GIMPLE trees have expressions already broken down into a three address form, which is needed for all later program transformations. GIMPLE trees are a relatively new development and they were added only in the 4.0 branch of the GCC. Data flow analysis brings the trees into a static single assignment (SSA) form, where each modification to a variable creates a new instance of that variable, so each variable is assigned a value exactly once.

Still, the most important intermediate representation is the register transfer language (RTL). RTL expressions - when printed as a debug information - look very much like Lisp expressions. These lists of expressions (called RTX for *register transfer expressions*) are used for most "classic" optimisations and eventually also for the actual code generation.[11]

Code generation works by simple pattern matching of instruction templates (which are - surprise - written in a Lisp-like form) with the register transfer lists.[12]

3.1.4. The GCC back end

To add a new target to the GCC a new configuration directory has to be added to the `gcc/config/` subdirectory and to the automake scripts.[13] The description of a target is split into three parts:

- `targetname.h`: This header file contains the definitions of many macros that define the compiler's behaviour. A complete list of all possible options is given in chapter 14 of [S+05]. Not every macro has to be defined for every architecture, but unfortunately, the documentation fails to tell which macros are absolutely needed and which may be omitted. As a rule of thumb, it is good practice to prepare a new header file by copying the documentation and the default values of most important macros from the internals manual.

- `targetname.md`: This is the machine description. In this file the semantics of every assembler instruction of the target machine should be defined. GCC defines a set of generic instructions which are used by the RTL intermediate representation.

[8] see [S+05], chapter 10.1
[9] [S+05], chapter 10.2
[10] see [Mer03]
[11] see [S+05], chapter 11
[12] [Par04] gives a very graphic example of how this process works in chapter 2
[13] [Par04], chapter 3.3, pg. 59ff provides a detailed description of the modifications to the make script that are necessary.

The machine description is then used to map each of these generic instructions to an assembler instruction, and it may also define side effects or constraints for each instruction.

There is a basic set of instructions that every back end has to define (mostly move, basic arithmetic and logic operations and jumps), but most other and more complex instructions may be left to the compiler to express through weaker operations. Then there is also the runtime library `libgcc` where unsupported instructions can be defined. These often include divisions and floating point operations.

- `targetname.c`: Many macro definitions can get rather complicated and are better implemented as separate functions. These functions handle the many details of the application binary interface (ABI) such as function entry and exit duties.

The runtime library `libgcc`

Not every processor can support the full feature set. It is the compiler's job to generate code for unsupported instruction patterns and in many cases this happens transparently to the programmer of the machine description. For some of these instructions it is more efficient to generate calls to library functions instead of inline code. These library functions form the compiler runtime library `libgcc`. The `libgcc` consists of two parts: one part has to be supplied by the author of the machine description (`libgcc1`) and a generic part `libgcc2`. Both are written in C, although some functions in `libgcc1` may be hardcoded in assembler.

`libgcc1` typically contains implementations of basic arithmetic and logic functions such as integer divisions.

Every target machine uses the same `libgcc2`, but the set of functions differs depending on whether there is a native implementation for a particular feature or not. In `libgcc2` there are arithmetic functions for non-native double precision integer modes but also initialisations routines that get called automatically by `main()` before the user program is run. To perform these initialisation tasks, the compiler inserts a call to a function called `__main()` right after the function header of the program's `main()` routine. This function manages an array of function pointers called `__CTORS__` that are to be filled in by the linker and point to the constructors of global C++ objects. Since the program cannot know whether it will be linked against a C++ module at a later time, this call has to be inserted into every C program as well.

The `libgcc` also contains emulation routines for basic floating operations. The implementation of the floating point functions is located in the file `gcc/config/fp-bit.c`.

Future plans

By the end of 2005 a discussion among GCC developers started about replacing the RTL back end with a technologically more advanced version. Currently there exists a proposal from the developers of the low level virtual machine (LLVM) project, some of which are now employed by Apple Computer, about merging the GCC front end with the

LLVM optimiser and code generators.[14] The LLVM project is featurewise comparable to the current GIMPLE/RTL back end, but is implemented in C++ and offers additional features such as compiling to a byte code (for later use with a just-in-time compiler, similar to the Java approach) and link-time optimisations.

From the discussion it seems as if consensus is that the RTL back end will be replaced over time but that it will happen gradually and not too fast, since it would be a too great effort to rewrite all back ends for all supported architectures.

3.1.5. Further documentation

A good reading before starting to port the GCC to a new architecture is [Nil00] combined with the official GCC Internals Manual ([S+05]), which serves well as a reference. While the official manual is quite complete, the meaning of some parts is easier to understand when compared to an actual implementation in one of the better documented back ends. One of the best back ends in that concern is the one for the Fujitsu FR-V family of processors, but the Axis CRIS back end is also very interesting to read, especially since it is the basis of [Nil00].

3.2. Port specific observations

Porting to specialised hardware can involve solving some tricky problems. Since it has to support almost 40 different architectures, the GCC is written to be easily retargetable, but it makes some assumptions about the target machine that can make porting a bit challenging. In his "Porting GCC for Dunces" Hans-Peter Nilsson writes: *"GCC is specifically aimed at CPU's with 32-bit general registers and byte-addressable memory"*.[15]

This might look intimidating if one sets out to port to a 24-bit word addressable machine, but previous ports have shown that something even harder is possible.[16]

This section contains an overview of the design decisions that were used to create the back end for On Demand's Control Processor. Also, there will be many examples of how to implement certain features in the machine description and a discussion of common fallacies that make the implementation harder than necessary.

3.2.1. Creating code for a 24-bit processor

One obvious problem was that there was no reference back end for a 24-bit architecture in the GCC's standard distribution. Almost all back ends used 8-bit bytes and a wordlength that is a power of two. The only exceptions were the TMS320C3x and TMS320C4x digital signal processors from Texas Instruments which used 32-bit bytes, and the pdp10 back end which used a hack to support 36-bit words and is no longer part of the standard distribution.

[14] [Lat05]
[15] [Nil00], pg. 17
[16] [KÖ5] shows that you can even port GCC to an 8 bit accumulator architecture that has only two registers

The difficulties with 24-bit words arise from GCC making assumptions like beeing able to calculate consecutive addresses by expressions of the form (*base_addr*+1) << *byteshift*, for example. But there are some strategies to minimise these problems that will be shown in this chapter.

The 24-bit words are also problematic in the compiled binary, as the GNU linker is not too happy about relocating addresses to targets that are not aligned at a power of two.

About Bytes and Units

If the smallest addressable unit of an architecture is a *byte*, that architecture is called *byte-addressable*. It is not recommended to try to port GCC to a word-addressable machine and this will cause a lot of problems that will require many of the GCC's internals to be changed.

The size of the smallest addressable unit of the target machine is called a *unit* in the GCC jargon, and can be defined through the `BITS_PER_UNIT` macro in the `targetname.h` file. The default size of a unit is 8 bits.

The GCC sources tend to use the terms `UNIT` and `BYTE` interchangeably which is very confusing and important to remember when browsing through the GCC sources.

Word-addressability

Most modern architectures are byte-addressable. There are several variants, though. The DEC Alpha, for example, can only address 32- and 64-bit words, but the addresses are still counted in 8-bit bytes (with the lowest 2 bits ignored), making it a byte-addressable machine in the GCC sense.

So how can the GCC generate code for a machine that counts addresses in steps of 24 bit and whose registers are 24 bit wide? The solution is this that that machine *is* actually byte-addressable, but one byte will be 24 bit long. This implies that all datatypes are at least 24 bit long as well. The common definition of a byte as an 8-bit entity predates the GNU compiler collection, where a byte is simply defined as the smallest accessible unit of a certain processor.

The gist of this dilemma is summed up by the following statements:

1. The GCC does not really distinguish between a Unit and a Byte.
2. The GCC can only generate code for byte-addressable machines.
3. The size of a *unit* = *byte* is not necessarily 8, although it should be a power of 2.

3.2.2. Machine Modes

The GCC distinguishes a number of *machine modes* that correspond to the data types in the programming language. On 32-bit architectures the machine modes default to the setup shown in figure 3.1.

Name	Length	Description
`QImode`	8 bit	Quarter-Integer
`HImode`	16 bit	Half-Integer
`SImode`	32 bit	Single Integer
`DImode`	64 bit	Double Integer
`SFmode`	32 bit	Single Precision Float
`DFmode`	64 bit	Double Precision Float
`BLKmode`	everything else	Arbitrary *blocks* of memory

Table 3.1.: Common machine modes and their default sizes on a 32-bit processor

Although the name suggests something different, especially `libgcc` depends on `QImode` to be defined, and to be the smallest available mode for the target processor. For this reason, the Control Processor back end defines `QImode` as the register-wide 24-bit mode and `HImode` as the double precision integer mode with a length of 48 bit.

Selecting the floating point modes is not as critical as the integer modes; on the Control Processor the `QFmode` and the `HFmode` and the floating point format of the back end for the TMS320C4x from Texas Instruments were chosen.

Floating point emulation

Even if the target machine is integer-only, most programming languages expect that floating point operations are available. On architectures that support a numerical co-processor, like the Intel 80386, this is solved by using an FPU simulator, that handles the illegal instruction interrupt. But for processors that do not support floating point instructions the compiler has to generate calls to an emulation library. The GCC comes with such a library, which is part of `libgcc`. The necessary functions are located in the `gcc/config/fp-bit.c` file, which is to be included in the *Makefile fragment* of the back end, `gcc/config/targetname/t-targetname`.

The author of the back end only needs to choose one of the supported storage formats, and the GCC will substitute library calls for all floating point operations. Also, an instruction pattern to copy floating point numbers needs to be defined.

Again, the floating point emulation library of the GCC makes some assumptions about the target machine that may be in conflict with platforms where the size of a byte is different from 8. The data type declarations expect the machine mode of `int` to be `SImode`, which is only true for byte-addressable 32-bit platforms. Another problem that may be encountered is that the generated floating point functions have the wrong mode-suffix; likely `sf`, for single-precision floats, where it should read `qf`, as it would make sense for a register-sized machine mode. This can be fixed by registering the floating point functions in the `TARGET_INIT_LIBFUNCS`, just as it is done with the integer functions of `libgcc1`, by a call to the `set_optab_libfunc()` function.

How to implement double-sized integers

Programming languages tend to offer many different data types to the programmer; processors sometimes don't. Large integers are necessary to hold the result of an integer multiplication, but also have uses for filesystem pointers and other things.

The GCC encourages the definition of data types that are unsupported by the target processor. Once the machine mode has been specified, it is sufficient to define the move operations using shorter instructions, and GCC will synthesise every other operation. It does so by splitting the data and storing it into consecutive registers. The resulting code will not be very efficient if the processor does not have a carry flag, though.

The back end for the On Demand Control Processor defines a 48-bit integer datatype that way.

3.2.3. Definition of instruction patterns

There are different kinds of instruction patterns that can be defined in the machine description. In the most straightforward case, a compiler-known instruction pattern would map directly to an assembler instruction on the target machine. For these cases, the `define_insn` patterns can be used. Examples for `define_insn` patterns are Figure 3.4 and 3.5. The `define_insn` expression takes five parameters:

1. The name of the instruction. In case of a compiler-known pattern (like `movqi`), it is used by the code generator to identify the instruction and its machine mode[17].
2. The RTL expression pattern. It defines the semantics of the instruction, and is also used by the code generator to match RTL expressions of unnamed expressions. RTL expressions consist of an operation whose operands may contain subexpressions. Each operand should have a postfix declaring its machine mode. The `match_operand` expression is used to refer to an operand of the instruction pattern.[18]
3. A condition. Conditions control the availability of an instruction on the current sub-target or in a specific pass of the compilation.
4. The assembler template. This is either a string that has to be processed and output to the Assembler, or a piece of C-code that returns the correct assembler template. It can also span several lines to provide different templates for every alternative specified by the constraints. See [S+05], chapter 13.6, for a more detailed explanation.
5. An optional vector of attributes. Attributes are used by the instruction scheduler to specify the type of an instruction (integer, jump), for instance.

Sometimes a standard pattern translates to a sequence of assembler instructions. In this case a `define_expand` pattern should be used. An example for this is the `pushqi` pattern on the Control Processor. During code generation, expand patterns are not

[17] GCC jargon for: data type
[18] A detailed explanation is given in [S+05], chapter 13.4

```
; PUSH
; sp = r62
;
(define_expand "pushqi"
  [(set (mem:QI (plus:QI (reg:QI 62) (const_int -1)))
        (match_operand:QI 0 "regimm_operand" "ri"))
   (set (reg:QI 62) (minus:QI (reg:QI 62) (const_int 1)))]
  ""
  ""
)
```

Figure 3.2.: How to synthesise a `push` pattern with atomic instructions

identified through their RTL pattern (the second parameter), but by their name. The RTL sequence that follows the name is then used to describe the operations that the instruction should be *expanded* to. They will then replace the pattern that had the instruction's name in the program and a new round of pattern matching will start.

Then, there is sometimes the necessity to generate different code for the same standard instruction depending on the circumstances. An example for this would be the `prologue` pattern. This is the other variant of *expander* definitions. In this variant the RTL template part is left empty and the RTLs are generated by C-code that is placed in the last parameter. This C function can deliberately stop the pattern matcher from trying to find another pattern for the current template via the `DONE;` macro. This is the way the function prologue instruction is implemented, for example. If the expander definition is only used to prepare or check some of the instruction's parameters, the `DONE;` macro can be omitted and the code generator will continue to look for a pattern that matches the one in the RTL template.

The branch instructions on the On Demand Control Processor are an example for the second variant of expander definitions. The problem is that the GCC expects the target processor to use a *condition code* (`CC`) register to save the result of a comparison instruction. On the Control Processor, conditions may be applied to any instruction, so the compare and branch instructions form a single more complex assembler line. In the approach taken, the pattern for the `cmpqi` operations just stores the parameters into global variables that are later used by the branch pattern which in fact only prepares the parameters for the real branch pattern that is specified in the RTL template as shown in figure 3.3.

A third kind of instruction patterns are *split* patterns. These patterns show the code generator how to replace a complex instruction with several simpler ones. Note that this is very similar to the expander definitions: In fact, both patterns are used for the same purpose, but they are evaluated in different stages of the compilation. Expander definitions are expanded very early, and are very flexible, whereas splits are performed in a later pass and impose stricter rules. For instance, it is unsafe to introduce new pseudo registers after the reload pass has completed. Splits behave almost like peephole

```
ilvy.md:

(define_expand "cmpqi"
  [(set (reg 60)
        (compare (match_operand:QI 0 "register_operand" "r")
                 (match_operand:QI 1 "regimm_operand"   "ri")))]
  ""
  {
    ilvy_cmp_op0 = operands[0];
    ilvy_cmp_op1 = operands[1];

    DONE;
})

(define_expand "bgeu"
  [(set (pc)
        (if_then_else (match_dup 1)
                      (label_ref (match_operand 0 "" ""))
                      (pc)))]
  ""
  "{ operands[1] = ilvy_emit_conditional_branch (GEU); }"
)

ilvy.c:

rtx ilvy_emit_conditional_branch (enum rtx_code code)
{
  /* Use the operands stored by the preceding cmp insn */
  rtx op0 = ilvy_cmp_op0, op1 = ilvy_cmp_op1;

  /* Zero the operands.  */
  ilvy_cmp_op0 = ilvy_cmp_op1 = NULL_RTX;

  /* Return the branch parameters */
  return gen_rtx_fmt_ee (code, Pmode, op0, op1);
}
```

Figure 3.3.: How to use an expander pattern to prepare arguments

optimisations. They also play an important role with delay slot filling.[19]

3.2.4. Instruction selection

As mentioned before, the GCC's instruction selection works by simple pattern matching, but it does provide mechanisms to control the selection process. One consequence of the pattern matching algorithm is that the order of the pattern definitions matters; if more than one pattern matches the RTL template one that was defined earlier will be used.

If no suitable pattern can be found, the expand pass tries to generalise the operation by using a mathematical equivalent expression or by splitting a complex operation into smaller sub-operations (an example for this would be a copy operation of an entity in BLKmode). If this does not work the expander tries to generate a call to a compiler-known function. A list of these functions is kept in the `optabs` which can be modified by the back end to let the compiler know of functions that are specified in the back-end-specific part of `libgcc`.

Unlike Burg-based code generators which use a tree-pattern matching algorithm, the GCC does not offer a too sophisticated cost model, though there is the `TARGET_RTX_COSTS` hook that lets the back end assign costs to specific RTL expressions.[20] The scheduler interprets the *cost* value that is assigned here as the latency of the instruction.

A more important role in the instruction selection process is occupied by *predicates* and *constraints*. While they both fullfill very similar tasks, which is to impose constraints on the possible usage of a specific instruction pattern, they are evaluated at different times. Their relationship could be described as predicates doing a rough preselection and constraints performing the fine tuning of the operands.

Predicates

A Predicate is used to describe the types of *operands* a certain instruction may take. Common predicates are `memory_operand` or `register_operand`. But of course the back end may define machine-specific predicates, like the often used `regimm_operand` on the On Demand Control Processor, which was used to denote operations that accept both a register or an immediate value. Figure 3.4 shows the use of predicates for the logical shift right operation on the Control Processor. While the destination and source operands need to be registers, the shift amount can either be given in a register or as an immediate value.

Constraints

Constraints allow for much more detail in the instruction selection process. As with the predicates, it is possible to have user defined constraints, too. A constraint is a short string that is used to define several variations of the same instruction. The example in figure 3.5 shows a *move* operation. Moves write to the first operand; thus the "=" in

[19] see [S+05], chapter 13
[20] see [S+05], chapter 14

```
(define_insn "lshrqi3"
  [(set (match_operand:QI 0 "register_operand"   "=r")
        (lshiftrt:QI
          (match_operand:QI 1 "register_operand" "r0")
          (match_operand:QI 2 "regimm_operand"   "ri")))]
  ""
  "%0 = %1 >>> %2;"
)
```

Figure 3.4.: The GCC instruction pattern for a logical right-shift on the Control Processor

the first line. The alternatives are seperated by commas: If the destination is a register, the source may be a register (r), a memory location (m) or an immediate value (i). Memory-memory moves are not permitted.

```
(define_insn "movqi"
  [(set (match_operand:QI 0 "nonimmediate_operand" "=r,  m")
        (match_operand:QI 1 "general_operand"      "rmi, ri"))]
  ""
  "%0 = %1;"
)
```

Figure 3.5.: The GCC instruction pattern for a copy operation the Control Processor

3.2.5. Instruction scheduling for VLIW slots

A relatively new addition to the GNU Compiler Collection is the inclusion of a *pipeline hazard generator*. To improve performance, modern processors come with many identical functional units that can process the instruction stream in parallel. On many processors this happens transparently, like the superscalar and out-of-order architectures, but some designs take a different approach and let the compiler do the work of assigning the instructions to functional units of the CPU.

An example for these are the very long instruction word (VLIW) architectures, like the Intel Itanium or the Fujitsu FR-V family of processors. The On Demand Control Processor also uses an explicit VLIW encoding.

The GCC provides a powerful description language for processor pipelines which integrates nicely with the rest of the machine description. This language allows the definition of the processor's functional units and the constraints that are imposed on the assignment of instructions to those units.

The instruction scheduler depends on an exact description of the processor's pipeline. It has to decide on whether an instruction may be issued at the current time, and how

to order the instructions so that idle cycles can be minimised.

To see if it is legal to issue an instruction, two kinds of constraints must be met:

1. Data dependency: All operands of an instruction must be available. This implies that an instruction may not depend on the result of any instruction that belongs to the same instruction bundle.

2. Instruction latency: Often the result of a more complicated calculation or a memory access will not be available immediately, but takes a fixed number of cycles to arrive. This is not an issue with the On Demand Control Processor.

To determine the availability of functional units at a certain time, the pipeline description is transformed into a deterministic finite-state automata (DFA) with the following mapping: States depict issue cycles with the assignment of functional units and transitions correspond to the possibility of issuing a certain instruction at this point in time.[21]

The machine description has to be extended to assign instruction class attributes to the particular instruction patterns, so the scheduler knows which processing units are going to be used, and what the latency of the instruction will be. The pipeline description is made up of regular expressions which are later used to create the DFA. The expressions describe how functional CPU units are allocated by each class of instructions. Figure 3.6 shows how an integer operation can execute in each of the units `slot0-slot3` and how the result will be available by the start of the next instruction bundle.

The statement declares that the "integer"-class units will be used and that the default latency is 1. The statement only applies to instructions that have the attribute "int" set. The final string parameter contains a regular expression that defines the possible allocation of the functional units. An expression like "`slot2+slot2`" means that for two consecutive cycles the unit "`slot2`" will be allocated. A "*" is shorthand for multiple applications of the "+" operator. Alternatives can be specified with the "|" operator.

```
(define_insn_reservation "integer" 1 (eq_attr "type" "int")
    "(slot0) | (slot1) | (slot2) | (slot 3)"
)
```

Figure 3.6.: Pipeline description: Reservation of CPU units by an integer instruction

The relationship between the functional units of the CPU would be declared as in figure 3.7.

The names of the units are declared in the `define_query_cpu_unit` statement. Two sets define the sequence in which the units may be allocated. The *presence set* of a given unit defines a set of slots that have to be already allocated to make the unit available. The *absence set* does the opposite and defines units that must not be allocated before.

The code that uses the DFA to perform the actual instruction scheduling is back end specific; luckily the back end for the FR-V family of processors contains a very general

[21] see [S+05], chapter 13.19.8

```
(define_query_cpu_unit "slot0, slot1, slot2, slot3")

(presence_set "slot1" "slot0")
(presence_set "slot2" "slot1")
(presence_set "slot3" "slot2")

(absence_set "slot0" "slot1 slot2 slot3")
(absence_set "slot1" "slot2 slot3")
(absence_set "slot2" "slot3")
```

Figure 3.7.: Pipeline description: Declaration of integer CPU units

implementation that can be reused with little modification. It even supports multiple instruction groups which is a nice feature in case the CPU is not built out of identical units.

The instruction scheduling algorithm

The bundling of instructions to VLIW packets is a process that happens in three steps[22]:

1. The GCC scheduling passes: The standard scheduler takes care of instruction latencies and tries to reorder the instructions in a way that pipeline stalls are minimised. This pass is only enabled at the optimisation levels `-O2`, `-O3` and `-Os`.

2. `TARGET_MACHINE_DEPENDENT_REORG`: This hook is called at all optimisation levels.[23] Here, the necessary `nop`s are inserted and the labels are aligned at the instruction word boundaries.

3. `TARGET_ASM_FUNCTION_PROLOGUE`: The actual bundling is performed in this hook. This hook was originally intended to output the assembler code for the function prologue, but has been superseded by the more powerful `prologue` instruction pattern. It is being hijacked for the bundling process because it is called just before a function is output to the assembler. The function prologue hook clears a flag for each instruction that should start a new bundle. It also reorders the instructions for processors that have different instruction classes associated with the instruction slots. Because this reordering process can destroy the meaning of the RTL stream and thus confuse the code generator, it can only be performed right before the assembly output happens.

Eventually, the `ASM_OUTPUT_OPCODE` hook interprets the bundling flag and adds the curly braces that indicate instruction bundles before printing the instructions to the assembler. While the DFA of the pipeline description is not used to assign the instructions to execution slots, all three passes use it to determine the correctness of a possible

[22] see the documentation in file *gcc/config/frv/frv.md* of [Fou05]
[23] see [S+05], chapter 14.29

combination of instructions in a VLIW bundle. The Control Processor back end uses a simplified version of the FRV back end's algorithm, because it has only one uniform class of integer execution units. The scheduler will add an instruction to the current bundle if the following conditions are satisfied:

- The total number of instructions in a bundle is < 4.
- The DFA allows to add the current instruction. This is done through the query interface of the pipeline description.
- There are no register or memory conflicts between the current instruction and all other instructions in the current bundle. For this check, internal data structures of the GCC can be used.

Unlike the FRV back end, the Control Processor back end currently does not reorder the instructions. For the reordering, the GCC's instruction scheduler is used.

3.2.6. Defining addressing modes

The decision wether a specific addressing mode is legal for the target processor is made by the `LEGITIMATE_ADDRESS_P` macro, which should be defined in the `targetname.h` file. This macro gets called twice for each address during the compilation. The first time it is invoked with an RTL expression that may still contain pseudo registers; the second time, a stricter check is needed, which means that the address will be used in that exact form as it is passed to `LEGITIMATE_ADDRESS_P`.

The GCC can and will use very complex addressing modes that may include side effects, like a post-increment of the address register - a feature that was included in an earlier version of the Control Processor. It will also try to generate memory indirect addresses and *address + offset* modes. The `LEGITIMATE_ADDRESS_P` may be used to restrict the size of the offset or displacement to 12 bit, as needed by the Control Processor.

3.2.7. Function prologue and epilogue

The `prologue` and `epilogue` instruction patterns are generated at the beginning and end of each function. There are also the older `TARGET_ASM_FUNCTION_PROLOGUE` and `TARGET_ASM_FUNCTION_EPILOGUE` macros that are called before a function is written to the assembler, but it is recommended to use the instruction patterns instead. However, the macros are still used as a trigger to perform instruction bundling.

The `prologue` and `epilogue` patterns are ideally written as expander definitions that generate the necessary code to set up the function stack frame and save the callee-saved registers. This code has to be supplied by the back end, as it depends heavily on the ABI.

In the Control Processor's back end the function prologue does the following things:

1. Push the old Frame Pointer to the stack.
2. Set the frame pointer (`fp`) to the value of the current stack pointer (`sp`)

3. Decrement the stack pointer by the size of the stack frame.

4. Save the return address on the stack.

5. Push every callee-saved register that will be used[24] by the function.

The epilogue performs the complimentary actions in reverse order. Eventually, it emits a `return` instruction.

3.2.8. Function calls

In order to implement call tables, virtual functions or function parameters it is necessary that the target processor supports a call to a non-constant address. The Control Processor does not have such an instruction. The `jump()` operation, on the other hand, would take the destination address in a register, but this approach would still leave the return address to be written to register 63. This leads to yet another problem: On the Control Processor, the program counter is not directly accessible either.

To solve this problem a label is emitted just before the jump to the function. Then, in order to get the program counter (PC) into `r63`, as the callee function expects it, a load-immediate of the label is issued. While the actual jump is made, `r63` is increased by one, to point to the address of the next instruction. Figure 3.8 shows the relevant piece of the machine description.

The `XEXP()` is just there to cope with an oddity of the assembler's syntax. Usually memory references are indicated by enclosing them in a `port[]` expression, but this does not apply to the targets of jump instruction. So the operand is de-referenced, to convert a `(mem:qi(reg:qi))` RTX into a `(reg:qi)` expression.

```
(define_insn "call"
   [(call (match_operand:QI 0 "memory_operand" "U")
          (match_operand:QI 1 "general_operand" ""))]
  ""
  {
    operands[1] = XEXP(operands[0], 0);
    if (REG_P(operands[1])) {
      return "{r63 = local_label%=; ; ; ; }\t"
          "local_label%=:"
           "\t{ r63 = r63 + 1; ; jump(%1); ; }";
    }
    return "jsr(r63, %1); // call\n";
  }
)
```

Figure 3.8.: How to fake access to the Program Counter

[24] the `regs_ever_live[regno]` predicate can be used to decide this

4. Defining an ABI

The application binary interface (ABI) defines the way that functions communicate with each other. It defines the calling conventions, the basic data type layout and the way a function stack frame is set up. The definition of an ABI should be the first step when porting a compiler. Often the ABI is already existent, but many times it has to be defined from scratch.

The ABI for the On Demand Control Processor was changed a few times during the design of the GCC back end. In the beginning the feasibility of 8-bit data types was evaluated. Since there is not much to gain by having the programming language support a non-native data type, this idea was discarded.

This chapter describes the final ABI for the On Demand Control Processor.

4.1. Data Types

The Control Processor has 64 registers with a wordlength of 24 bits. The ILVY assembly language allows for 24- and 12-bit immediate values. See [Win05] for more information. The standard C data types are defined as follows:

C Type	Length
`int`	24 bit
`short int`	24 bit
`long int`	48 bit
`char`	24 bit
`float`	24 bit
`double`	48 bit

Table 4.1.: Common C data types and their respective sizes on the Control Processor

Characters are represented by 24 bit. While this is inefficient, text processing is not the key field of application of the Control Processor. The Control Processor has no support for floating point operations; they have to be emulated in software.

4.2. Memory Layout

The data memory can be addressed in steps of 24 bits. Thus, all data structures should be aligned on 24 bit boundaries. Bytewise addressing modes have to be emulated by the compiler.

Variables are stored at address `port[0]` and upwards, the stack is located at `port[4095]` and grows downwards.

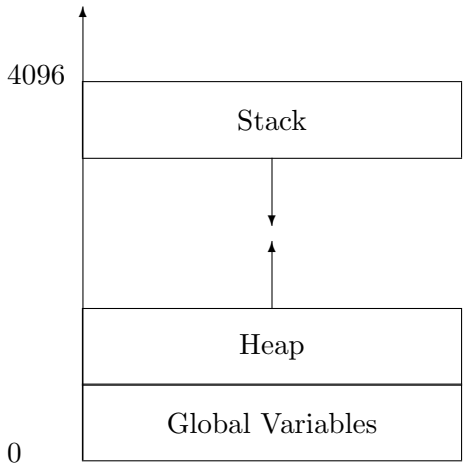

Figure 4.1.: Data memory layout for the On Demand Control Processor

4.3. Register Usage

Name	Usage	Comments
`r0`	function return value	
`r1-r6`	argument registers	
`r7-r14`	callee-saved registers	
`r15-r60`	caller-saved temporary registers	
`r61`	frame pointer	(or callee-saved register)
`r62`	stack pointer	(grows downwards)
`r63`	return address	

Table 4.2.: The purpose of the Control Processor's registers

Register `r0` holds the return value of a function. If the return value is larger than the size of `r0`, a pointer to a location on the stack is passed. The first six arguments are passed in registers `r1-r6`.

Starting at `r7`, there are 8 *callee-saved* registers. If a function wants to use these registers, it has to save them onto the stack and restore their original values before returning to the caller function.

The bulk of the registers, `r15-r60` are *caller-saved* registers. They may get overwritten when a function is called. It is good practice to always have more caller-saved registers than callee-saved registers. To fully utilise the many registers of the Control Processor, an interprocedural register allocation mechanism would be of advantage. Unfortunately,

the GCC does not yet support this. Function inlining is one way to increase the register usage, but it also increases the code size significantly.

The *frame pointer* is stored in `r61`. It points to the beginning of the function stack frame and is used in the function epilogue to restore the original layout of the stack on function return. It may be omitted in some cases[1], in which it is used as an additional callee-saved register.

The *stack pointer* marks the current bottom end of the stack, and is stored in `r61`.

Finally, the return address for the current function is passed in `r63`. This is the address of the instruction after the last function call.

4.4. Function Stack Frame

The first 6 function arguments are passed in `r1-r6`, all other parameters are pushed to the stack. In functions with variable numbers of arguments (such as ",..." parameters in C) these parameters are passed on the stack to ensure array-like behavior.[2]

Local Variables reside in temporary registers where possible, and are spilled to the stack when necessary.

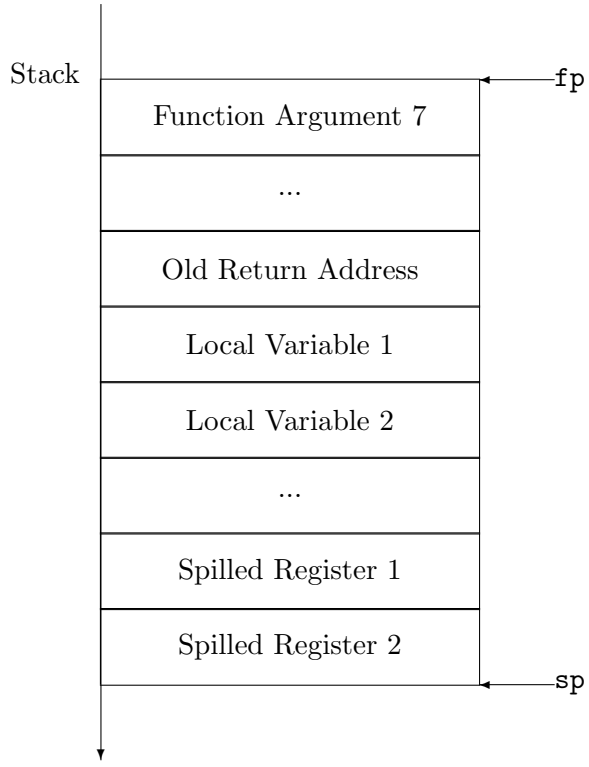

Figure 4.2.: layout of the Function Stack Frame

[1] see the documentation of the GCC compiler switch `-fomit-framepointer`
[2] see [Nil00] pg. 66f

5. GNU Binutils

The GNU Binutils is a collection of utilities that operate on binary files. They include tools that are essential for building programs, like the GNU Assembler (`gas`) and the GNU Linker (`ld`).

Subdirectory	Description
`bfd`	The BFD library - the basis of all the Binutils
`binutils`	A collection of various binary utilities, such as `objdump`
`gas`	The GNU Assembler
`gprof`	The GNU Profiler
`ld`	The GNU Linker
`libiberty`	A tiny subset of the C standard library
`opcodes`	The disassembler library

Figure 5.1.: The components of the GNU Binutils

The Binutils can be ported independently from the compiler and ideally two people or teams could work on the compiler and the binutils at the same time.

In this chapter, an introduction into the most important libraries and tools of the GNU Binutils is given. Also, there will be short description of the modifications necessary for supporting a new target architecture.

5.1. Manipulating object files with `libbfd`

The tools that form the GNU Binutils have in common that they read or write binary object files at some point. The functions to perform binary file I/O and the management of the necessary symbol tables are concentrated in the BFD[1] library, `libbfd`. This library consists of a front end that the programs talk to and a lot of back ends for all sensible combinations of object formats and architectures.

The BFD library supports many different object file formats, such as a.out, b.out, the common object file format (COFF) and the executable and linking format (ELF). If a new architecture is to be defined, it is suggested that the ELF format is chosen. It has shown to be the most flexible and widest supported format.[2]

The `libbfd` has to live a double life, because it is distributed with two different packages, the GNU Binutils and the GNU Debugger (GDB). This fact complicates porting, because there are two versions of `libbfd` that need to be maintained. So it is strongly

[1] This acronym has been retro-fitted to stand for Binary File Descriptor

[2] see [Cha03], chapter 1

suggested to use a version of GDB that was released closely after the version of the Binutils. Otherwise it would be difficult to patch two incompatible versions of the same library. To make things even more confusing, the GDB and Binutils projects seem to have release cycles of about a year, but tend to release alternating every 6 months. This remark is also valid for the disassembler library, `libopcode` that is also shared by binutils and GDB.

5.1.1. Sections

A binary file is divided into different sections that are reserved for machine instructions (".code", ".text"), initialised data (".data") and unitialised data (".bss"). The exact amount and location of sections is different from file format to file format, but the above three should be supported by all formats. An application may have - depending on the object format - more than one of each section.[3]

However, during the assembly some more section names are introduced; they are used to indicate unresolved references. They are called "const" sections and are the *absolute* section, the *undefined* section, the *common* section and the *indirect* section. These sections are global and values in there may not be changed by the application. However, they may vanish in the linking process.[4]

5.1.2. Symbols

For every label[5] that is used, the assembler makes an entry into the symbol table. The symbol table is a list of all labels defined in the current file, and the location they point to in the context of the file. Therefore the assembler has to pass a list of every symbol to the BFD library.

Symbols have several flags attached to them.[6]

- BSF_GLOBAL: This defines whether a symbol should be visible from other modules. A public global variable or the name of a library function should have this flag.
- BSF_LOCAL: Local implies that the name is not unique and may be shared by multiple labels from different files.
- BSF_UNKNOWN: This is for symbols that are referenced in the current file, but are defined somewhere else.

Some information about symbols is carried by the section a symbol belongs to: For example, if a symbol is an uninitialised global variable, its section should always point to `com_section_ptr`, which is defined by `libbfd`. The common keyword is used in the

[3] For a very technical overview, see [Hau98]

[4] see [Fou06a], file `bfd-in.h`

[5] For an assembler, there is no difference between a label and a variable. Thus the term "label" is used throughout this section.

[6] see [Cha03], chapter 2.7, pg. 39

assembler to mark uninitialised global variables that are located in the "common" data pool. It will show up as section "*COM*" after the assembly - if looked at with objdump -x - and the variables will be stored in the ".bss" section after linking.

```
.data
var list_ptr = list; // Symbol "list_ptr", Reloc for "list"
.code
public main: // public symbol "main"
{
    r1 = port[list_ptr + 1]; // Reloc for "list_ptr", addend 1
    jsr(r63, free); // Reloc for "free"
    ;
    ;
}
```

Figure 5.2.: Symbols versus Relocations

5.1.3. Relocations

The counterpart of the symbol table is called a *relocation*. Relocations exist to solve the following problem: The assembler has to translate all references to symbols into numeric addresses, but because it assembles only one file at a time, it cannot know the final numeric value of an address. It is the *linker*'s job to go through all symbol references and relocate them to their final values. Therefore the assembler has to prepare a table containing every symbolic reference within the code and the initialised data. Figure 5.2 shows an example of assembler code that generates symbol table entries as well as relocations.

Typically, a back end defines many different kinds of relocations. Data might be initialised to be a simple pointer (word-sized absolute relocation), but can also point to the member of a data structure; in this case an offset has to be added to the value of the relocation (this is called the *addend* in the BFD jargon). A symbolic reference in the code could be either absolute, like a library call, but also relative to the program counter (PC-relative), like branches to a very close target on some machines.

In order to support relocations, libbfd has to be extended with a configuration file for the new architecture, which is to be named [format]-[cputype].c. For each type of relocation the architecture wants to support, a HOWTO-macro needs to be defined in this file. The HOWTO-macro explains the linker how to find the exact location and perform the relocation. This is not trivial, since often only part of a binary encoded instruction has to be replaced or modified with a new value.

5.1.4. Porting the BFD library

Porting libbfd involves creating two files: a very short CPU definition that is located in cpu-[cputype].c, which contains the bfd_arch_info_type record for the new architec-

ture, and the definition of the architectures relocations and other features, [format]-[cputype].c.

It is best to decide on the types of relocation the architecture needs to support first. Then, mostly the HOWTO macro and a lookup mechanism to translate the generic relocation types into the correct mechanism, as described by the HOWTO macro.

The HOWTO macro is insufficient for 24-bit architectures, because it expects the linker to be able to convert relocated addresses to the target's native format through a binary rightshift. The best solution to this problem would be to add a *divisor* field to the macro and extend the bfd_perform_relocation() function to divide every address by this field. This way, arbitrary bytelengths could be supported. The interim implementation on the Control Processor moves this task to the elf loader of the simulator, that has to convert each address before executing a program.

Another issue on the Control Processor is that the way libbfd handles relocations is not ideally suited for harvard architectures. Consider the following piece of assembler code:

```
{
    r1 = data;
    r2 = data_len;
    r3 = 1;
    r4 = compare_function;
}
{
    jsr(r63, qsort);
    ;
    ;
    ;
}
```

Figure 5.3.: Pointers into two different memories.

The obvious problem is that the assembler cannot know where an address is pointing to. On machines where code and data memory have the same bytelength this is not a problem. But on the Control Processor, the code memory is addressed in steps of 1 instruction word, which is much longer than the 24 bits of a data word. For this reason the assembler will install an *unknown* relocation for each symbol which is then replaced - done by the special function supplied to the HOWTO macro - with a *code* respectively *data* relocation, depending on whether the symbol that is referenced lies in the .text or in the .data or the .bss section.

5.1.5. Debugging

In the process of debugging newly defined relocation types, the objdump is an essential tool. Combined with a libopcode disassembler library it can show the symbol tables,

assembler instructions and relocations of an object file or a linked executable.

The `objdump` tool is part of the GNU Binutils distribution and heavily based on the BFD library.

5.2. The Assembler `gas`

The GNU Assembler (gas) is really a collection of assemblers that are primarily meant to be used to work with the output of the GCC. If there is a system assembler on a particular architecture gas will usually emulate its behaviour. Still the GNU assembler has a very distinct syntax and is not easily adjusted to support more complicated assembler languages like the one used for the On Demand Control Processor. But for its main purpose, which is to be a quick translator for the output of the GCC its design seems to be appropriate.[7]

5.2.1. Porting gas

To add a new assembler to the collection of GNU assemblers, the parser for the assembly language has to be rewritten. Basically this consists of a routine that translates one line of assembler code into some internal `emit_*` calls, that tell gas how to look up the binary instruction codes (opcodes) and where to place relocation entries.

5.2.2. A comparison with the native assembler

As mentioned before, the native assembler for the On Demand Control Processor is different from the GNU Assembler in that it supports a more complicated assembler dialect and is written in C++. It also makes extensive use of the `lex` and `yacc` tools to parse the assembly language.

This approach was taken because the Control Processor was originally to be programmed only in assembler and thus had to offer a user-friendly language. It also was designed to be easily extendible. With the availability of a compiler, however, these demands have shifted; and performance is now more important over readability.

For this reason, there is a certain possibility that future versions of the Control Processor will use a `gas` based assembler.

5.3. The Linker `ld`

The GNU Linker is the last tool that is invoked when compiling a program. Its job is to combine all the different object files and libraries into the final executable. It further also relocates the data and code sections and ties up all symbol references. To do these tasks, `ld` makes extensive use of the BFD library that is an integral part of the GNU Binutils.

[7] [EFf02], chapter 1

GNU ld gives the user full control over the linking process; this is done through the *Linker Command Language* which is a superset of the AT&T Link Editor Command Language syntax. The GNU linker is also known for its ability to continue on errors and to provide more useful debugging information than older linkers.[8]

5.3.1. The Emulation

The different combinations of linker scripts and the default behaviour of the linker for a specific target are called *emulations*. A linker target usually has to support several variations, which can be controlled by the user via command line switches.

Each variation has its own default linker script. The linker scripts for a certain target are generated from a general definition at build time. This general definition consists of the linker script embedded in a shell script; this shell script has to emit the final linker script and will be called with different settings of various shell variables.[9]

With this mechanism it is possible to generate the scripts for all emulations a target must support.

5.3.2. The Linker Script

The linking process is totally scriptable, which is an important feature, since different architectures need very different layouts of executable files. The GNU linker allows to define a default linking script for the target architecture, but the user may also provide a different script for more flexibility.

Linker Scripts control the final layout of the named sections in the executable file. Each section can be described by its name and two addresses. There is the virtual memory address (VMA) that is the address at which the section will be located when the final executable will be run. Then there is also the load memory address (LMA), which is the address at which the section will be loaded. These two addresses may point to the same location, but a section could be loaded into ROM and the be copied into RAM at runtime. This mechanism makes it possible to initialise global variables in ROM-able code.[10]

5.3.3. Porting ld

To port the GNU linker to a new target architecture there is not much to be done. This is mostly because ld is merely a front end to functionality found in the BFD library. In most cases it is sufficient to supply a new linker script. Figure 5.4 shows the linker script for the On Demand Control Processor.

The Control Processor is a Harvard Architecture which means that it has separate memories and address spaces for data and code. This is usual practice with many digital signal processors (DSP) that run static code off a specialised instruction memory.

[8] [CT04], chapter 1
[9] see [BCTD00], chapter 2
[10] see [CT04], chapter 3

Unfortunately this bears a little annoyance, because the linker will always complain about overlapping code and data sections. The only solution for this problem is to manually disable the warning in the `elf.c` file of the BFD library.

The Linker Script of the On Demand Control Processor is a very simple one, which only loads the individual sections one after another and takes care of the memory layout described in the application binary interface.

```
SECTIONS {
        . = 0x0; /* let the code start at address zero */
        .text : { *(.text) }
        . = 0x0; /* data starts at address zero, too */
        .data : { *(.data) }
        .bss : { *(.bss) }
        _end = .; /* _end marks the end of the .bss section */
}
```

Figure 5.4.: The linker script for the On Demand Control Processor

The `_end` pointer is a special symbol used by the C library *Newlib*. It marks the end of the data section and thus the beginning of the heap memory. It is used by the `_sbrk()` function that is called by `malloc()` to request another block of memory.

6. The GNU Debugger

Work on the GNU Debugger began shortly after the GCC project was started. Again, the main author was Richard Stallman, but soon the project would be maintained by Cygnus Solutions which was founded by John Gilmore and was bought by Red Hat in 2000.

The GNU Debugger has excellent support for C and C++ but also some other languages, knows about as many target platforms as are supported by the BFD library and supports remote and simulator based debugging. It is perhaps the most used debugger to date.[1]

6.1. The structure of GDB

The GNU Debugger consists of three components.[2]

1. The user interface.
2. The symbol management part, with binary interpretation performed by the BFD library, the debug symbol interpreter and also support for the source languages.
3. The target side, which includes execution control and memory manipulation routines.

A debugging session may either run *native*, where GDB is compiled to run on the same machine as the debugged program, *remote*, where it communicates with the target machine over a network or *simulated* where the program is run in a simulator that is invoked by GDB.[3]

In the case of an embedded system, choosing one of the last two options is more sensible. Especially for automated testing (of the GCC, for instance) the simulator target has many advantages.

6.2. Using GDB

In order to use the GNU Debugger with an integrated simulator, it is necessary to switch the target for the session. This can be done by issuing a "`target sim`" command to the debugger. This will tell GDB to use the target platform's simulator.

[1] see [She06], chapter 1
[2] see [GS05], chapter 2
[3] see [SPS+05]

If the debugger is to be used interactively; to find bugs and inspect unexpected behaviour of programs, there exist a number of very good graphical front ends to GDB, such as the Data Display Debugger (`ddd`).[4]

6.3. Porting the GNU Debugger

The GNU Debugger depends - as most of the utilities discussed herein - on the GNU BFD library and the `libopcode` disassembler library. Unfortunately, the maintainers of GDB decided to ship GDB with its own copy of `libbfd` which is usually a snapshot of another time in development than the one that comes with the GNU Binutils.

After the modifications of `libbfd` were carefully transcribed into GDB's version of `libbfd`, the *target vector* has to be written:

The target vector is the data structure that defines the interface between the GDB and the debugged process or the simulator.[5] The GDB needs very detailed information about the ABI, as it has to find local variables inside the stack frame, for instance. This information should go into a file called `target-tdep.c` in the `gdb/` directory of the sources.

The other important part - at least for an embedded processor - is the *simulator interface*. The `sim/` subdirectory contains interfaces for all target machines that have a simulator defined. These interfaces perform execution control (like the loading of a program and single-stepping through it), target manipulation (like reading and writing to registers and variables on the simulated machine) but also provide a *callback interface* that can be used to perform tasks like `printf()` on the host machine, so it is possible to test almost any program on targets that have very limited capabilities.

The simulator interface has a number of function calls that control the simulator and the callback library:

- `sim_open(), sim_close()`: For the Control Processor, these functions initialise and terminate an instance of the simulator.
- `sim_load()`: This is the function to load a program into the simulator. The exact functionality varies a little depending on the type of simulator (process or a hardware). On the Control Processor (hardware) simulator, the program is loaded into memory, and the program counter is reset to the `_start` symbol.
- `sim_read(), sim_write()`: These functions allow GDB to read and write to memory locations on the simulated hardware.
- `sim_fetch_register(), sim_store_register()`: Through these functions, the debugger can inspect the values of the simulated CPU's registers, and also modify their contents.

[4] http://www.gnu.org/software/ddd/
[5] see [GS05], chapter 10

- `sim_info()`: This is a debugging function that asks the simulator to print any statistics it has collected.
- `sim_resume()`: This call instructs the simulator to either perform a single step or to resume the simulation. It also allows the debugger to send an event (hardware interrupt or operating system signal) to the simulated program.
- `sim_stop()`: The *stop* is an asynchronous event to stop the simulation immediately.
- `sim_stop_reason()`: Through this interface, the debugger can query the reason why the simulation has stopped. If the simulation was stopped by any signal, the type of that signal is reported, too.

The creation of breakpoints is handled by a callback in the `target-tdep.c` file that registers them with the simulator.

7. The New C runtime library

Many embedded systems make use of the `newlib` C runtime library. This is a collection of library functions that were assembled by Cygnus Support and came from different sources. They were released under a BSD-type license. It is the only part of the system that is not officially a GNU project.[1] The `newlib` implements a full C standard library, including an IEEE floating point library. It also supports several operating system specific calls, such as `getpid()`, `gettimeoftheday()` or `exit()` that are implemented as *stubs* wich means that their functionality has to be provided by the user. Alternatively the `newlib` also supports several embedded operating systems such as uC/OS or Linux.

The `newlib` is made up of three parts:

- `libc:` This is the C runtime library. It contains functions such as `malloc()`, `printf()` and `strcpy()`. To reduce the size of the library, the amount of supported features can be controlled by configuration variables. There is an integer-only version of `printf()`, for example.
- `libm:` This is the mathematical part of the runtime library. It implements functions like `sinf()` or `sqrt()` that are not part of `libgcc2`.
- `libgloss:` Routines that are close to hardware or the operating system are contained in this library. It is the only part of `newlib` that needs to be modified when adding a new architecture.

Compiling the `newlib` can be a good benchmark for testing the quality of a new compiler back end. Inside the libraries are many complicated initialised data structures and pointer arithmetic expressions that can prove challenging to get right.

7.1. Defining system specific issues

The GNU low level operating system support library (`libgloss`) is the part of `newlib` that manages the program startup and the bridge between standard library and operating system (or the debugger, for that matter). These functions contain both hardware and system specific parts that once belonged to `newlib` (where they were located in the `libc/sys/` subdirectory) and are no located in a separate library.

The `libgloss` has two main parts, the `crt0.o`, handling the startup procedure of every program and the system call interface.

[1] see [Gat06]

7.1.1. System startup: `crt0.o`

The C Runtime Zero (`crt0.o`) gets linked at address 0 of the `.text` section. It is responsible for bootstrapping the application. This process covers the following steps:

- Initialisation of the hardware: For the On Demand Control Processor this means setting up the stack pointer to the highest memory location available. Most targets also fill the `.bss` section with zeros.
- Call `main()`: This includes setting up the program arguments. The first thing that `main()` will do is to call `__main()` which calls all elements of the function pointer array `__CTORS__` in order to initialise all global C++ objects. The `__CTORS__` array has to be filled by the linker.
- Shutdown: This should include a jump to `exit()`, to signalise the hardware or the simulator that the program has terminated.

The `crt0.o` is usually written directly in assembler; the file should be called `crt0.s`.[2]

7.1.2. System Calls

The `newlib` target has to provide an interface to several *system calls* that can be implemented as calls to an embedded operating system or to interact directly with the hardware. Of course, not all of these system calls make sense for every platform, so some are usually defined to simply fail silently.

For testing purposes it can be useful to pass these calls through the simulator interface of the GNU Debugger (`gdb`) which can be taught to support host-native system calls when run in the simulator target. Through this mechanism it is possible to run arbitrary programs on platforms that do not even have support for the most basic I/O.

7.2. Porting newlib

Porting the `newlib` basically means to define the system part of the library, which is `libgloss` and the `/sys` subdirectory of `libc`. Also, the `newlib` offers control of various parameters through compile-time flags. These optional features include floating-point support, reentrant versions of the system calls and the implementation of the `malloc()` function.

To port `newlib` some modifications to the linker script have to be made, too. The most important addendum is the definition of the `_end` variable, that marks the end of the `.bss` section and thus the beginning of the heap. The `_sbrk()` function, which is responsible to allocate another chunk of heap for later use by `malloc()`, needs this variable.

[2] see [Sav95], chapter 3

8. Generating a complete toolchain

The process of creating a complete compiler toolchain for a new target is rather long and non-linear. This chapter tries to give an overview of the tasks necessary to create a full build system and how the different tools work together.

1. The first thing that is needed is the ABI. It defines the way that programs will behave on the new target and is the specification for compiler, linker and runtime system.

2. Then, in parallel, work on the assembler and the compiler back end can be started. Porting the GNU assembler involves porting the BFD library, because the assembler has to generate relocatable object files.

3. Porting GCC involves defining a lot of macros that define the ABI and the machine itself, and writing the machine description. Once the compiler itself can be compiled, the newly built compiler will try to compile its support libraries (which is `libgcc1` and `libgcc2`). For this to work, the assembler for the target machine must be ready, but also some of the Binutils for creating library archives, like `ar`.

   For testing purposes it is useful to write a little assembler stub that allows to create executables that can be simulated without the runtime library.

4. Before work on the runtime library can begin, it is necessary to prepare the linker scripts for the GNU linker.

5. The next step would be to port the runtime library `newlib`. If `newlib` should compile, it will be possible to compile a complete program for the first time.

6. Meanwhile, one can also start porting the GNU Debugger (`gdb`) to the new target. The Debugger needs either real hardware, in case of a native version, or an instruction-level simulator.

7. After these steps the final phase of quality assurance can be started. It is important not to underestimate the effort of this last step. The testsuite of the GCC will be introduced in the next section.

8.1. Testing with DejaGNU

The GNU Compiler Collection comes with a very large testsuite that is based on the DejaGNU testing framework. DejaGNU is written in *Expect*, a dialect of the TCL

language. The testsuite contains more than a 20000 testcases that have historically shown to often break with modifications of the compiler.

The testsuite is loosely structured into different groups:

- C-torture: These testcases were submitted together with bugreports, and make up the largest part of the testsuite. They are divided into *compilation* and *execution* tests.
- gcc.dg: These tests ensure certain compiler behaviour and usually test specific features.
- gcc.target: In here are target specific tests. This is the place for back end authors to write testcases for their new target,
- g++.dg, g77, etc.: Then there are also language specific testcases that are written in C++ or Fortran. Above tests were all written in C.

The `test_installed` script in the `contrib/` directory of GCC is a nice tool to automatically run the tests for the newly build compiler. But be warned that a run of the testsuite can take several hours to complete!

8.1.1. Porting DejaGNU

In order to support a new target platform - or *board*, as it is called in the DejaGNU jargon - it is necessary to add a description of that platform to the DejaGNU sources and reinstall the framework.

The necessary file should be called `basebords/targetname-sim.exp`. Again, it is best to copy the definition from a related platform. The file has to define several Expect-functions that control the loading of a program and the spawning of a new simulator process. It also contains the definitions of several variables that inform the DejaGNU driver of the features of the particular baseboard.

For the On Demand Control Processor, the baseboard was defined to directly spawn a simulator process, without going over the GDB-bridge.

9. Evaluation of the final product

In this final chapter, some aspects of the code created by the compiler back end for the Control Processor are presented. This chapter does not try to be a comprehensive benchmark of the compiler; it aims to be an overview about what the new GCC back end can do (at the time of writing) and where there is still room for improvement. Finding a measurement for the code quality is not an easy task, especially because it has a very restricted featureset that makes comparing it to another platform difficult.

In this chapter, some demonstration programs and their performance on the instruction level simulator are presented, with attention to various optimisation parameters and the instruction scheduler.

9.1. Benchmarks

9.1.1. Generating pseudo-random numbers

The first program serves as a demonstration of the features of the compiler. This program uses a *lagged fibonacci sequence* that is stored in a short buffer to generate pseudo-random numbers. This example was chosen because the implementation of the algorithm is really short and secondly does not use any multiplication at all. The method was devised in 1958 by G. J. Mitchell and D. P. Moore[1] and works by calculating the sequence $f_n = f_{n-24} + f_{n-55} \pmod m$. The implementation suggested in [Knu98] uses an even shorter buffer with only 55 memory cells, but since conditional execution has yet to be implemented in the compiler, it would generate lots of unnecessary jumps, which is why the slightly less memory-efficient implementation with 64-word buffer was chosen. Figure A.1 shows the source code of the program in C.

For the test, the following parameters were varied:

1. Optimisation flags: The program was compiled without optimisation (`-O0`), optimisation for speed (`-O3`) which does not include loop unrolling, and optimisation for size (`-Os`).

2. Number of instruction bundles: The compiler for the Control Processor allows to specify a special parameter (`-mslots=`n) which instructs the instruction bundling pass to pack only up to n instructions into a bundle. The values used were 1 (also the default value for compilation without optimisation), 2, 3 and 4 (the default when optimisation is turned on).

[1] The method is discussed in detail in [Knu98] , chapter 3.2.2, formula (7)

3. Number of memory ports: A second back-end-specific parameter (`-mports=`n) can be used to specify the number of memory accesses that are allowed in one instruction bundle. This parameter is interesting because building a quadruple-ported memory is much more expensive compared to a single- or dual-ported memory.

These are the parameters that were monitored:

1. Execution speed - Total Cycles: The performance was measured in total number of cycles used by the instruction level simulator including application startup time.

2. Jump penalty: This is the number of jump penalty wait cycles. Each jump on the Control Processor costs two additional cycles.

3. Code size: Code size is a very important parameter for the Control Processor. Aside from memory constraints, a future version may include a cached program memory and a smaller code size would yield less cache misses which would introduce unnecessary wait cycles.

4. Slot efficiency: This parameter is very interesting for the design of VLIW processors. The tradeoff between the number of parallel functional units and code size is to be considered, which is measured by the average number of CPU slots that were used per instruction word. The number given in the *dynamic slots* column refers to the instructions that are actually executed, so an often used instruction is counted multiple times. The *static slots* column, on the other hand, shows the slot utilisation of all instruction bundles, disregarding how often they are executed.

The assembler code the compiler generated for the `rand()` function - optimised for program size and with instruction bundling turned on - is listed in Figure A.2. The instructions to store register 63 (the return address) on the stack could actually be omitted, which would save another cycle.

The following tables contain the results of running 1000 iterations of the algorithm in `rand.c` under varying conditions. As the base case, shown in table 9.1, the unoptimised program[2] is executed, with instruction bundling disabled.

Slots	Total Cycles	Jump Penalty	Dyn. Slots	Stat. Slots	Code Size
1	56,097	6,132	1.0	1.0	784 Bytes

Table 9.1.: running the rand.c program with `-O0`

When optimisations are disabled, only one instruction is issued per instruction word. This is necessary to facilitate debugging. Since conditional jumps still use two instruction slots, the average number of slots is slightly higher than 1.

In table 9.2, the program has been optimised for size[3] and with the instruction bundling forced to use 1-4 slots. This is the suggested default set of parameters to

[2] `ilvy-odm-elf-gcc -mslots=1 -O0 -o rand.O0.1.exe rand.c`
[3] `ilvy-odm-elf-gcc -mslots=n -Os -o rand.Os.1.exe rand.c`

compile a program for the Control Processor with. This table shows that number of slots have a direct influence on both codesize as well as performance. Due to data dependency issues, the utilisation of the instruction slots does not grow significantly from 3 slots to 4 slots. The number of memory ports, however, does not have a significant influence on the result, since the generated code does not access more than two memory locations per instruction bundle anyway.

Slots	Total Cycles	Jump Penalty	Dyn. Slots	Stat. Slots	Code Size
1	33,508	6,124	1.0	1.0	464 Bytes
2	22,380	6,124	1.75	1.68	296 Bytes
3	20,265	6,124	2.01	2.09	240 Bytes
4	17,263	6,126	2.55	2.52	200 Bytes

Table 9.2.: running the rand.c program with `-Os`

If the same program is optimised for speed[4] the performance almost doubles compared to the size-optmised version above.

Slots	Total Cycles	Jump Penalty	Dyn. Slots	Stat. Slots	Code Size
1	21,427	2,116	1.0	1.0	768 Bytes
2	15,306	2,116	1.54	1.62	496 Bytes
3	14,245	2,116	1.68	2.11	384 Bytes
4	10,244	2,116	2.51	2.54	320 Bytes

Table 9.3.: running the rand.c program with `-O3`

This is shown in figure 9.3. The strong reduction of jump penalty cycles comes from GCC actually inlining the `rand()` function, thus eliminating roughly 1000 calls and returns at 3 cycles each.

9.1.2. Encrypting Data

The *Blowfish* algorithm is a cryptographic algorithm that was designed by Bruce Schneier in 1993. It is a keyed, symmetric block cipher that is known to be very fast, although it has a relatively big memory footprint of at least 1 kilobyte read-only data.[5]

The following test measures the effort to encrypt and then decrypt a block of 1024 64-bit words.[6] Again, the tests were performed with different optimisation settings and a varying number of slots and memory ports.

[4] `ilvy-odm-elf-gcc -mslots=n -O3 -o rand.O3.1.exe rand.c`

[5] see [Blo06] and [Sch94]

[6] The Blowfish algorithm is inherently 32 bit. To yield a comparable result, this benchmark was performed on a specially modified Control Processor with 32-bit registers. Porting GCC to the new bytelength was done in about half an hour, since it only involves changing a few constants.

Opt.	Slots	Total Cycles	Jump Penalty	Dyn. Slots	Stat. Slots	Code Size
-O0	1	5,029,722	330,344	1.0	1.0	4,416 Bytes
-Os	1	2,512,218	313,640	1.0	1.0	2,608 Bytes
	2	1,881,441	313,640	1.44	1.62	1,656 Bytes
	3	1,475,463	313,640	1.94	2.06	1,312 Bytes
	4	1,453,815	313,640	1.98	2.15	1,256 Bytes
-O3	1	1,688,164	103,512	1.0	1.0	5,072 Bytes
	2	1,160,183	103,512	1.56	1.60	3,272 Bytes
	3	1,025,265	103,512	1.78	1.87	2,800 Bytes
	4	769,725	103,512	2.47	2.28	2,320 Bytes

Table 9.4.: running the `blowfish` program with the Control Processor simulator

9.1.3. Performing error correction

The *Viterbi* algorithm is a method to perform error correction on digital signals that were transmitted over a very noisy channel. The algorithm uses dynamic programming to find the most likely sequence (the Viterbi-path) of hidden states that trigger a sequence of observed events. A common use is the decoding of the convolutional code used by GSM phones or wireless LAN networks.[7]

In this benchmark the viterbi algorithm was used to decode short block of noisy data.

Opt.	Slots	Total Cycles	Jump Penalty	Dyn. Slots	Stat. Slots	Code Size
-O0	1	6,273,330	708,614	1.0	1.0	4,928 Bytes
-Os	1	2,380,106	486,854	1.0	1.0	2,056 Bytes
	2	1,611,472	454,014	1.88	1.68	1,280 Bytes
	3	1,397,096	454,014	2.3	2.03	1,064 Bytes
	4	1,294,141	454,018	2.58	2.24	968 Bytes
-O3	1	2,554,382	614,498	1.0	1.0	2,104 Bytes
	2	1,607,822	485,186	1.91	1.69	1,296 Bytes
	3	1,557,529	614,498	2.29	1.97	1,128 Bytes
	4	1,306,859	485,188	2.61	2.23	992 Bytes

Table 9.5.: running the `viterbi` program with the Control Processor simulator

9.1.4. Discrete Cosine Transform

A *discrete cosine transform* (DCT) is an integral part of lossy singal encodings. Using this operation, a time-discrete signal can be transformed into frequency space. The DCT is closely related to the Fourier transform, but uses only real numbers.[8]

[7] see [Vit06]
[8] see [DCT06]

Opt.	Slots	Total Cycles	Jump Penalty	Dyn. Slots	Stat. Slots	Code Size
-O0	1	10,977	1,152	1.0	1.0	8,296 Bytes
-Os	1	4,667	992	1.0	1.0	3,728 Bytes
	2	3,398	992	1.71	1.61	2,384 Bytes
	3	3,108	992	1.95	1.83	2,096 Bytes
	4	3,200	1,120	2.11	2.10	1,832 Bytes
-O3	1	3,214	556	1.0	1.0	8,904 Bytes
	2	2,340	556	1.7	1.50	6,112 Bytes
	3	2,073	556	1.99	1.71	5,368 Bytes
	4	2,256	684	2.09	1.77	5,192 Bytes

Table 9.6.: running a discrete cosine transform Control Processor simulator

In this example a discrete signal of 32 values is transformed to the frequency space. Figure 9.6 shows a suboptimal behaviour of the register allocator at the highest optimisation level. What happens is that in order to increase the utilisation of the slots, more working copies of values are generated, which unfortunately decreases overall performance a little.[9]

9.2. Performance improvements with VLIW bundling

To generate the diagrams in this section, the data gathered by the benchmarks above was normalised using the single-slot variant of the Control Processor as baseline. All tests were made at the highest optimisation level GCC offers (`-O3`).

In figure 9.1, the number of simulated CPU cycles for each benchmark is plotted on the y-axis. The four bars for each benchmark represent the speed gained through adding more slots to an instruction bundle. Figure 9.2 shows the efficiency of the various instruction bundle sizes. This plot is not normalised; it shows the absolute number of instructions per (executed) bundle. This corresponds the *dynamic slots* column in the benchmark tables. Figure 9.3 shows the reduction of code size as more instructions are packed into a VLIW bundle. Again, these values are normalised with the single-instruction configuration as baseline.

The plots show that the current way of instruction bundling yields about a twofold increase in performance as well a twofold reduction of the number instruction words per program.

9.3. Varying the number of memory ports

In its default configuration, the Control Processor has one memory port per instruction slot. Since the cost of a memory with multiple ports is higher than a single- or dual-port version, it is important to analyse the effect of reducing the number of memory ports.

[9] For more information, see also chapter 11.1

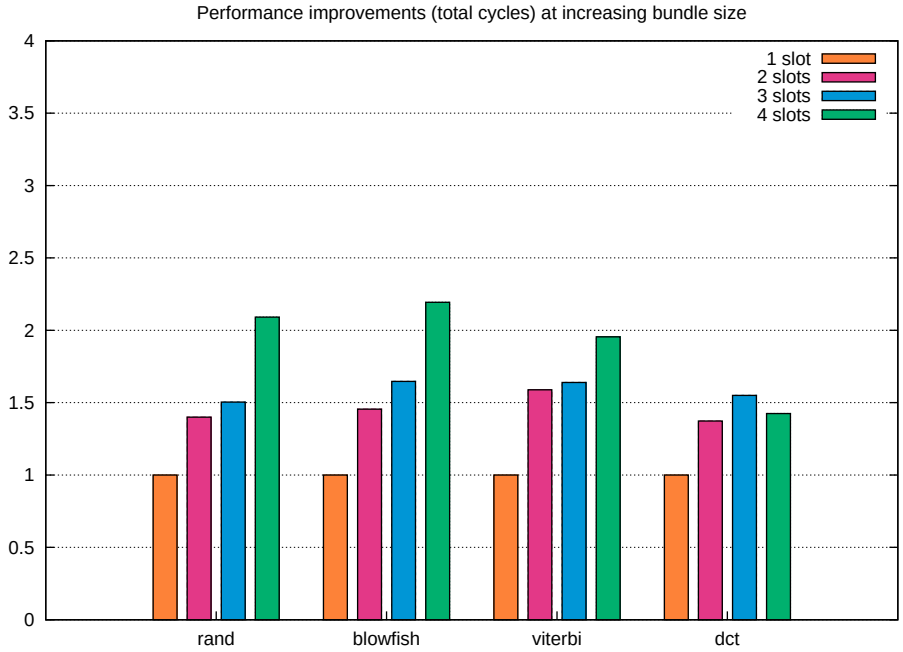

Figure 9.1.: Execution time in correlation to VLIW bundle size

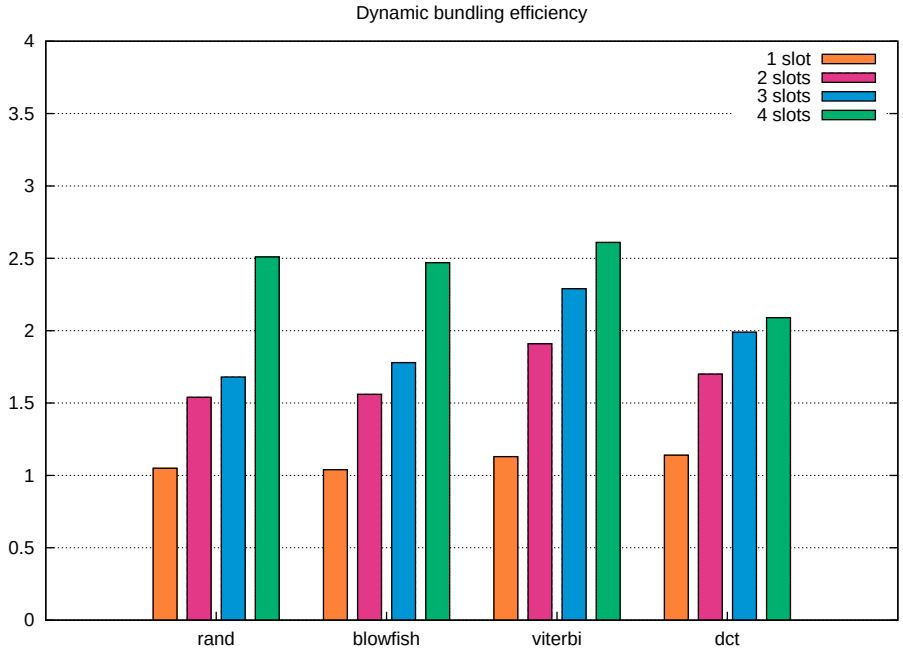

Figure 9.2.: Average number of instructions per VLIW bundle at different sizes

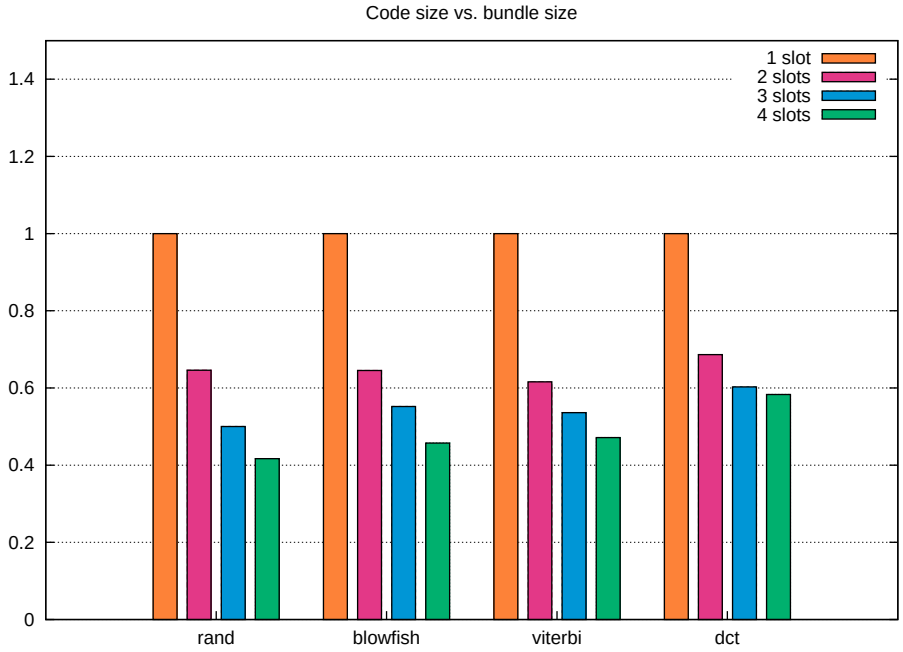

Figure 9.3.: Total number of instruction words in correlation to VLIW bundle size

In this section, the previous tests were repeated with a varying number of memory ports and different optimisation settings.

Opt.	Ports	Total Cycles	Jump Penalty	Dyn. Slots	Stat. Slots	Code Size
-Os	1	19,264	6,124	2.16	2.16	232 Bytes
	2	17,263	6,126	2.55	2.52	200 Bytes
	3	17,263	6,126	2.55	2.52	200 Bytes
	4	17,263	6,126	2.55	2.52	200 Bytes
-O3	1	11,246	2,116	2.23	2.25	360 Bytes
	2	10,244	2,116	2.51	2.54	320 Bytes
	3	10,244	2,116	2.51	2.54	320 Bytes
	4	10,244	2,116	2.51	2.54	320 Bytes

Table 9.7.: running the `rand` program with different memory configurations

During one iteration, the `rand` benchmark accesses only two memory cells of the ring-buffer containing the last 64 generated values. This is the reason why it does not profit from adding more than 2 memory ports, as shown in table 9.7.

The performance of the `blowfish` benchmark is not really affected by the number of memory ports. Table 9.8 shows only a minor improvements.

Opt.	Ports	Total Cycles	Jump Penalty	Dyn. Slots	Stat. Slots	Code Size
-Os	1	1,531,025	313,640	1.85	1.79	1,504 Bytes
	2	1,457,932	313,640	1.97	2.09	1,296 Bytes
	3	1,457,931	313,640	1.97	2.10	1,288 Bytes
	4	1,453,815	313,640	1.98	2.15	1,256 Bytes
-O3	1	822,249	103,512	2.29	2.08	2,536 Bytes
	2	769,725	103,512	2.47	2.24	2,352 Bytes
	3	769,725	103,512	2.47	2.28	2,320 Bytes
	4	769,725	103,512	2.47	2.28	2,320 Bytes

Table 9.8.: running the `blowfish` program with different memory configurations

Opt.	Ports	Total Cycles	Jump Penalty	Dyn. Slots	Stat. Slots	Code Size
-Os	1	1,297,312	454,018	2.57	2.08	1,040 Bytes
	2	1,295,197	454,018	2.57	2.21	984 Bytes
	3	1,295,197	454,018	2.57	2.21	984 Bytes
	4	1,294,141	454,018	2.58	2.24	968 Bytes
-O3	1	1,309,503	485,188	2.6	2.09	1,056 Bytes
	2	1,307,915	485,188	2.61	2.20	1,008 Bytes
	3	1,307,387	485,188	2.61	2.22	1,000 Bytes
	4	1,306,859	485,188	2.61	2.23	992 Bytes

Table 9.9.: running the `viterbi` program with different memory configurations

Similar results exist for the `viterbi` and `dct` benchmarks. For completeness, they are given in table 9.9 and 9.10.

Opt.	Ports	Total Cycles	Jump Penalty	Dyn. Slots	Stat. Slots	Code Size
-Os	1	3,563	1,120	1.79	1.38	2,760 Bytes
	2	3,304	1,120	2.01	1.81	2,120 Bytes
	3	3,261	1,120	2.05	1.91	2,016 Bytes
	4	3,200	1,120	2.11	2.10	1,832 Bytes
-O3	1	2,555	684	1.76	1.45	6,312 Bytes
	2	2,327	684	2	1.66	5,528 Bytes
	3	2,315	684	2.02	1.69	5,448 Bytes
	4	2,256	684	2.09	1.77	5,192 Bytes

Table 9.10.: running the `dct` program with different memory configurations

10. Related Work

The classic work about porting the GCC is definitely "Using and Porting GNU CC" by Richard Stallman. This book forms the basis for the GCC Internals Manual which is a more up-to-date version of the same text, available online and released together with each version of GCC. This manual is a must-read for everyone who plans to work with the GCC. [S+05] is structured like a reference manual and covers both language front ends, the back end and the optimiser. Not all topics are described equally detailed and some chapters are merely stubs but it still is the most complete description of the internals of the GCC.

A more practical approach was taken by "Porting GCC for Dunces" by Hans-Peter Nilsson which aims to be an introduction to writing back ends for the GCC. Nilsson is the author of the back ends for the Axis CRIS and D. E. Knuth's hypothetical MMIX processor. In [Nil00], the development of the CRIS back end is described step by step. The CRIS is a 32-bit processor with 16 general-purpose registers that was designed with the GCC in mind. The document describes the intrinsics of the CRIS architecture and in a second part a selection of the GCC's target macros and their functions. The focus of [Nil00] is clearly on the definition of the target macros which are described in [Sta98], too.

In "Free Development Environment for Bus Coupling Units of the European Installation Bus" Martin Kögler describes how he adopted the GNU toolchain to the M68HC05 microcontrollers from Freescale (former Motorola). In this work he had to deal with harsh memory and register constraints, as the M68HC05 has only two 8-bit registers and only a few tens of bytes of data memory. To conserve memory, new machine mode sizes in the range of 1 to 8 Bytes were introduced. Also, to ease code generation for GCC, a set of virtual registers was defined. This work differs from the others in that it does not only concentrate on the GCC back end but also deals with the other parts of the toolchain. Due to the fact that the targeted microcontroller is 8-bit and very constrained the compiler faces a very different set of problems than the ones described in this work.

VLIW processors are targeted specifically by Jan Parthey's "Porting the GCCBackend to a VLIW-Architecture". He describes the GCC port for the TMS320-C6000 digital signal processors (C6x) made by Texas Instruments. This diploma thesis contains a very high-level overview of the compilation process and explains many instruction patterns of the C6x back end. The work skips one important topic, though, as VLIW packing is not actually implemented.

This gap is filled by Adrian Strätling in "Optimizing the GCC Suite for a VLIW Architecture". In this follow-up work to [Par04], the GCC port for the TI TMS320-C6000 DSP is enhanced with instruction scheduling, conditional execution and VLIW

packing. The work focuses on optimising the GCC back end to generate better code and shows many ways to improve the performance of VLIW code. Strätling dives into more advanced topics than there were implemented in the Control Processor back end.

The generation of ELF object files using the BFD library is explained in [Gre97]. This article explains how the BFD library was used to construct a tool that transforms the output of the Stanford compiler framework SUIF to generic ELF object files that could be loaded into a simulator for an in-development DSP processor.

11. Conclusion

This work represents the result of a little over six months of work. During this period a complete development environment for the On Demand Control Processor was created, almost entirely by adopting the freely available GNU toolchain. The resulting environment consists of a near-complete C99 compiler, an assembler that creates ELF object files, a linker, a runtime library and a basic debugger that interfaces to a simulator.

The final compiler shows that GCC is fit for VLIW architectures, as it creates parallel instruction bundles that exploit the capabilities of the target processor to a high degree. The experimental evaluation also shows that some kind of global register allocation would be an advantage, since GCC has difficulties with using the many registers the Control Processor offers.

Regarding the unusual byte-length of the Control Processor, it seems that many of the GNU tools (especially the BFD library) would profit from a generalisation of the assumptions they make on a CPU's characteristics. But it was also shown that it is still possible - but not necessarily easy - to create a 24-bit tools from their current versions. GCC itself does not have a problem with byte-lengths that are not a power of two as long as the target processor is still byte-adressable.

The performance of the generated code is a good start; the efficiency is increased by the parallelism the VLIW slots offer, still there are some features left waiting to be implemented, such as conditional execution and a better register allocation mechanism.

11.1. Directions for future work

Although the presented development environment is already in a very usable shape, there is still room for improvement. There are interesting features that are yet to be implemented; this section will point out some of them.

Variable-length Arguments: There are some functions in the C language that support a variable number of arguments. The best-known example would be the `printf()` function. The GCC supplies a standard implementation of this functionality that unfortunately only works on machines that pass function arguments on the stack.[1] In order for the `<varargs.h>` implementation to work, a handful of macros have to be defined.

This feature has been left out because standard I/O has not been too much of an issue at the time of writing.

[1] see [S+05], chapter 14.11

The BFD library: The BFD library (integral component of the GNU Binutils and GDB) should be extended to properly support wordsizes that are not a power of 2. The current implementation of 24-bit relocations on the On Demand Control Processor is not ideal and could use a better integration with the `libbfd`.

The GNU Debugger: Unfortunately the port of the GNU Debugger is all but complete; almost all functions that are necessary to support interactive debugging have not been implemented. The reason for this was that it was possible to use the debugging functionality of the standalone instruction-level simulator.

Leaf Functions: The GCC allows further optimisations to be made if it knows that a certain function will be a *leaf function*. Leaf functions are functions that do not call any other functions. For that reason, they can omit prologue functionality like saving the return address register. They also do not need to bother saving caller-saved registers, naturally.[2]

These features have not been implemented because they probably only cause a small performance boost and the Control Processor has a sufficient number of registers.

Conditional Execution: The On Demand Control Processor allows any instruction to be preceded by a condition. Using such mechanisms, the compiler could prefix a short block of instructions with the same condition and thus eliminate a rather expensive jump. The GCC allows for conditional move and add operations (`movmodecc` and `addmodecc`[3]) and there is also the `define_cond_exec`[4] instruction pattern that can be used for this purpose.

Interprocedural Register Allocation: This is one more interesting project. The Control Processor offers a lot more registers than are usually needed by a single function. Thus, it would make sense to implement a way to share a pool of registers by several functions that are closely related. Unfortunately interprocedural register allocation is a feature yet to be implemented into GCC.

[2] see [S+05], chapter 14.7.4
[3] see [S+05], chapter 13.9
[4] see [S+05], chapter 13.20

A. Program Listings

A.1. The `rand.c` benchmark

```
/* rand.c
 * A lagged-fibonacci-sequence pseudo-random number generator
 * that works without using multiplications;
 * see D. Knuth, TAOCP2: Seminumerical Algorithms, chapter 3.2.2 (7)
 */

unsigned int fibs[64];
int n = 0;

void init_fibs(void) {
    /* initialise the array with arbitrary start values, not all even */
    for (n = 0; n < 55; n++)
        fibs[n] = 1 << n + n;
}

int rand() {
    ++n;
    return fibs[n & 63] = fibs[(n-24) & 63] + fibs[(n-55) & 63];
}

int main() {
    int i;
    init_fibs();

    for (i = 0; i < 1000; i++)
        rand();
    return rand();
}
```

Figure A.1.: The source code of the pseudo-random number generator.

```
public rand:
{
    r16 = port[n]; // movqi
    r18 = fibs; // movqi
    port[r62 + (-1)] = r63; // movqi
    r62 = r62 - 1; // subqi3
}{
    r17 = r16 + 1; // addqi3
    r16 = r16 + (-23); // addqi3
    ;
    ;
}{
    r19 = r17; // movqi
    r17 = r16 & 63; //andqi3
    r16 = r16 + (-31); // addqi3
    ;
}{
    r20 = r17; // movqi
    r16 = r16 & 63; //andqi3
    port[n] = r19; // movqi
    r19 = r19 & 63; //andqi3
}{
    r20 = r20 + r18; // addqi3
    r16 = r16 + r18; // addqi3
    r19 = r19 + r18; // addqi3
    ;
}{
    r17 = port[r20]; // movqi
    r18 = port[r16]; // movqi
    ;
    ;
}{
    r17 = r17 + r18; // addqi3
    ;
    ;
    ;
}{
    port[r19] = r17; // movqi
     r0 = r17; // movqi
    ;
    ;
}{
    r63 = port[r62]; // movqi
    r62 = r62 + 1; // addqi3
    ;
    ;
}{
    ret(r63);
    ; ; ;
}
```

Figure A.2.: The generated assembler code for the `rand()` function.

Bibliography

[BCTD00] Per Bothner, Steve Chamberlain, Ian Lance Taylor, and DJ Delorie. *A guide to the internals of the GNU linker*, 1992-2000.

[Blo06] Blowfish (cipher). `http://en.wikipedia.org/wiki/Blowfish_(cipher)`, 2006.

[Cha03] Steve Chamberlain. *libbfd*, 1991-2003.

[CT04] Steve Chamberlain and Ian Lance Taylor. *Using ld*, 1991-2004.

[DCT06] Discrete cosine transform. `http://en.wikipedia.org/wiki/Discrete_cosine_transform`, 2006.

[EFf02] Dean Elsner, Jay Fenlason, and friends. *Using as*, 1991-2002.

[Fou05] The Free Software Foundation. The gcc source code. `http://gcc.gnu.org/`, 2005.

[Fou06a] Free Software Foundation. The bfd library source code. `http://www.gnu.org/software/binutils` and `http://www.gnu.org/software/gdb`, 2006.

[Fou06b] The Free Software Foundation. *Using the GNU Compiler Collection (GCC)*, 2006.

[Gat06] Bill Gatliff. Porting and using newlib in embedded systems. `http://billgatliff.com/drupal/node/25`, 2006.

[Gre97] Jack Greenbaum. Generating object files directly from suif/machsuif using gnu libbfd.a. `http://suif.stanford.edu/suifconf/suifconf2/papers/15.ps`, 1997.

[GS05] John Gilmore and Stan Shebs. *GDB Internals*, 1990-2005.

[Hau98] Michael L. Haungs. Extending sim286 to the intel386 architecture with 32-bit processing and elf binary input. `http://www.cs.ucdavis.edu/~haungs/paper/paper.html`, 1998.

[KÖ5] Martin Kögler. Free development environment for bus coupling units of the european installation bus. Master's thesis, TU Wien, 2005.

[Knu98] Donald E. Knuth. *The Art of Computer Programming*, volume 2. Addison-Wesley, 1998.

[Lat05] Chris Lattner. Llvm/gcc integration proposal. `http://gcc.gnu.org/ml/gcc/2005-11/msg00888.html`, 2005.

[Mer03] J. Merrill. Generic and gimple: A new tree representation for entire functions. *First Annual GCC Developers Summit*, 2003.

[Neu05] Karl Neumann. The ilvy assembler source code, 2005.

[Nil00] Hans-Peter Nilsson. Porting gcc for dunces, 2000.

[Ogr05] Julia Ogris. The ilvy simulator source code, 2005.

[Par04] Jan Parthey. Porting the gcc-backend to a vliw-architecture. Master's thesis, Chemnitz University of Technology, 2004.

[PH98] David A. Patterson and John L. Hennessy. *Computer Organisation and Design*. Morgan Kaufmann, 1998.

[S+05] Richard M. Stallman et al. Gcc internals manual. `http://gcc.gnu.org/onlinedocs/gccint/`, 2005.

[Sav95] Robert Savoye. Embed with gnu. `http://gcc.gnu.org/onlinedocs/gccint/`, 1995.

[Sch94] Bruce Schneier. Description of a new variable-length key, 64-bit block cipher (blowfish). In *Cambridge Security Workshop Proceedings (December 1993)*, pages 191–204. Springer-Verlag, 1994.

[She06] Stan Shebs. Gdb: An open source debugger for embedded development. `http://www.redhat.com/support/wpapers/cygnus/cygnus_gdb/`, 2006.

[SPS+05] Richard Stallman, Roland Pesch, Stan Shebs, et al. *GDB Internals*, 1988-2005.

[Sta98] Richard M. Stallman. *Using and Porting GNU CC*. Free Software Foundation, 1998.

[Sta06] Richard M. Stallman. The gnu project. `http://www.gnu.org/gnu/thegnuproject.html`, 2006.

[Vit06] Viterbi algorithm. `http://en.wikipedia.org/wiki/Viterbi_algorithm`, 2006.

[Win05] Siegfried Winterheller. *Control Processor Programmer's Guide*. On Demand Microelectronics, 2005.