

Porting GCC

Bryan Richter
ECS 199, Winter Quarter '05
UC Davis

April 1, 2005

This report documents the work undertaken by the author to create a new port for GCC. The target architecture is the TMS320C54*x* line of DSP:s produced by Texas Instruments. This was an independent research project (ECS 199) in UC Davis' Dept. of Computer Science, and was mentored by Chris Nitta.

This project was motivated by the Neuros¹, a portable music player. The Neuros' developers have embraced Open Source ideals, and have contributed most of their code to the community. Unfortunately, it is difficult for community members to contribute to the project, since no free compiler exists for Neuros' system. Porting GCC to the 'C54*x* family, to which the Neuros' chip belongs, would alleviate this problem.

Porting GCC is no small task. As a consequence, the stated goals of this project were to make as much headway as possible and document the work done. These goals were met, as the existence of this report might suggest.

As expected, the amount of work unfinished dwarfs the amount of work completed. In fact, most of the completed work has been simply figuring out what must be done. The results of this work have been a better understanding of GCC internals, an outline of the porting process, and an introduction to DSP theory.

1 GCC

The GNU Compiler Collection² is the flagship of the Free Software movement. Due to the work of countless contributors from across the globe, GCC can produce assembly code for dozens of systems. It can also compile four languages—C, Ada, Fortran, and Java. GCC's flexibility is what allows GNU/Linux operating systems to operate on nearly any hardware.

GCC achieves its flexibility with a modular design. It is broken into three parts: the front-end lexes and parses the input code of a program, and converts it to a language-independent representation. The program then passes to a language- and system-independent 'middle-end', which does the majority of the optimizations. Finally it is passed to a back-end that does final optimizations and generates assembly code.

An understanding of the middle-end is necessary to make a port, but its discussion is deferred to the references [1]. This project is mainly concerned with the back-end.

1.1 GCC Back-End Description

The back-end depends on three files that give a description of the target machine. In this project, those files will be named `c54x.h`, `c54x.c`, and `c54x.md`. These files serve two purposes. One is to define things such as endianness, address size, number of registers, and target-specific command-line parameters. The other purpose is to describe the machine's instructions and instruction attributes. It also describes machine-specific optimizations.

1.1.1 Machine Definitions

Two of the three files, `c54x.h` and `c54x.c`, are used to define the machine. These are standard C files. The definitions are C macros that all reside in the header file. The `.c` file is used in cases where a macro would

¹<http://www.neurosaudio.com>

²<http://gcc.gnu.org>

be large or unwieldy. For example, if `#define BIG_AND_SCARY` proves to be a troublesome macro, it would be best declared as `#define BIG_AND_SCARY big_and_scary()`, with `big_and_scary()` defined in `c54x.c`.

Hundreds of macros exist in GCC documentation, and a sampling of other ports shows 200–400 macros actually used. These macros depend on more than just the machine. If compliance with another compiler is desired, the definitions must match the ABI used by that compiler. An ABI (Application Binary Interface) describes how things such as variable-length parameters are implemented. The macros will also be subject to change; some of them might need to be tinkered with to improve optimization as the port matures. Clearly, much time will be invested in this section of the port.

1.1.2 Instructions and Attributes

The third file, `c54x.md`, describes machine instructions and attributes. Unlike the other two files, this one is written in RTL. RTL is the intermediate representation used by the middle-end, wherein the entire logical structure of a compiled program is represented as a list of RTL expressions (RTX:s). An example of RTX is `(plus:SI (const_int 2) (const_int 2))`, which represents `(int)(2+2)`. The patterns that match a particular RTX will include the assembly instructions that cause the processor to do the action specified by the RTX.

The third file also defines instruction attributes. Their use and purpose has not been explored yet, however.

The `.md` file is no shorter than the header file. There is a long list of specific patterns that must be defined. Also, the RTX:s can be quite complex. Besides generating assembly, they can also describe how to split one RTX into two or more, or how to join multiple RTX:s into one.

1.2 Writing the Back-End

Here are the steps required for creating the three machine description files. Note that this is a preliminary list. Since this project still has a long way to go, this list might be modified as more information is uncovered.

The first step is to fill out the header file. This is the simplest step for a couple of reasons. First, it is all done in C (it is expected that anyone undertaking a GCC port is proficient in C). Second, most of the needed information will be available in the target system's documentation. One could step through the macros listed in the GCC Internals manual and fill them out, one-by-one.

The second step is defining the required RTX:s. This will require a familiarity with RTL. Also, instruction attributes will need to be studied and designed in concert with this step. While that makes the task more daunting, the good news is that once this step is complete, it should be possible to compile GCC. Unfortunately, all preceding work must be done blindly, without the aid of a debugger. From this point on, however, the project can be modified incrementally with rich debugging support (GNU's GDB works particularly well with GCC, as one might expect).

There are a number of things that must be done after this step. The order in which they should be done is unclear at this point, so they will be listed randomly. A linker and assembler must be configured to create correct executable files, so programs can be loaded on the device. More RTX:s must be written to make machine-specific optimizations. A profiling and feedback system should be created to allow the proof of the effectiveness of tweaks and optimizations. Finally, it is certain that more tasks will present themselves as the project progresses.

2 Progress

As mentioned previously, the majority of the completed work has been reading and research. This research allowed the understanding of GCC that is outlined above, and was vital to the rest of the work.

After the reading, the header file received a fair amount of work. About 50 macros are currently defined. While many of them were quickly defined, a few of the macros required some thought and planning. An example is the set of macros concerned with register classes, which required defining a hierarchy of register class subsets.

3 Future Work

There is plenty of work that could build off this project. To start, of course, the port must be completed. At the author's current level of expertise, the next few steps of this process are quite clear. Beyond these is certainly an equal or greater amount of work, however. Further, once the port 'works', a lot more things could be done to expand its usefulness.

The next thing that should be done, of course, is finish the macros. As explained above, this is probably the best task to approach first. The steps following this are all outlined above in the subsection "Writing the Back-End". Specifically, the named instructions listed in GCC documentation must be defined, and the instruction attributes need to be fleshed out. Finally, there will be plenty of bug fixing and tweaking, which is a major factor in any project.

As also mentioned above, once GCC emits correct assembly, more work will have to be done to get the assembly to run on a device. This means configuring an assembler and (more importantly) a linker to set up data and code in a meaningful way for the system.

Beyond this are many possible projects. Some are directly involved with the port. Optimizing the port is one of these. This would be a very lengthy project, involving a lot of profiling, tweaking, and understanding of DSP technologies. Another project would be expanding the port to handle all the 'C54x family members. Yet another would be adding support for specific systems (DSP, external memory, I/O, etc.) such as the Neuros which inspired this project.

Other possible expansions would be porting or developing free software for systems that employ 'C54x:s. Porting GCC serves the express purpose of opening these chips to the Open Source community. This project would be best served by its use in other Open Source projects.

References

- [1] *GCC Internals Manual*. <http://gcc.gnu.org/onlinedocs/gccint/>, Mar. 20 2005. Free Software Foundation, Inc.
- [2] *TMS320C54x DSP Reference Set, Volume 1*. Texas Instruments, Inc. Fort Worth, Texas, 2001.
- [3] *TMS320C54x DSP Reference Set, Volume 2*. Texas Instruments, Inc. Fort Worth, Texas, 2001.