Appeared in the Proceedings of GSPx'04, Santa Clara, September 2004

Porting GCC to the TMS320-C6000 DSP Architecture

Jan Parthey
Community4you GmbH
Annaberger Straße 240

09125 Chemnitz
Germany
Phone: +49 371 5347 242
jan.parthey@informatik.tu-chemnitz.de

Robert Baumgartl
Chemnitz University of Technology
Department of Computer Science
Real-Time Research Group
09107 Chemnitz
Germany
Phone: +49 371 531 1592
robert.baumgartl@informatik.tu-chemnitz.de

Abstract

This paper describes our efforts towards the implementation of a C compiler for the Texas Instruments TMS320C6x DSP architecture based on the GNU Compiler Collection GCC. We give a detailed motivation and introduce GCC basics. Following, we motivate important design decisions made during the work and demonstrate the current state of the project by looking closely at some code generated by our compiler. The paper is finished with a critical evaluation and an outlook onto future project stages.

1 Introduction

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

Currently, a number of projects at Chemnitz University evolve around the TMS320C6x DSP by Texas Instruments [1]. To our knowledge, the only C compiler currently available for this architecture is a commercial one, which is offered by the manufacturer.

There are a number of reasons why this situation is not satisfactory: Firstly, with the current TI compiler, the interface for embedding hand-optimized Assembler code into C code is far from optimal, mainly due to lacking means for explicit register allocation in such Assembler blocks. By using unallocated registers, the compiler user would be at risk of overwriting register values used by the enclosing C code for holding temporaries and local variables. Only whole functions can therefore safely be implemented in Assembler code. However, the resulting performance is not optimal, because frequent function calls to small routines often require unacceptably many CPU cycles. Given that only time-critical parts of the algorithms shall be hand-coded, it is thus virtually impossible, with the TI compiler, to speed up DSP programs using inline Assembler code. A second reason is the fact that open source software projects are customarily optimized for being compiled with GCC and often cannot easily be compiled with the TI compiler.

Having target support for the C6x implemented in GCC would help to bring a variety of Linux software to the C6x platform. Since uClinux could eventually be run as operating system, porting should be relatively inexpensive [6]. Another option is the implementation of RTAI, the Real-Time Application Interface [2].

Because of GCC's sophisticated interfacing mechanisms to inline Assembler code, this would also solve the first problem mentioned above. Another advantage would be the support for all input languages accepted by GCC, which would include C++, Fortran, and others. Finally, GCC is distributed under the GPL, which would allow tailoring the C6x target to special optimization requirements, if this is needed.

The rest of the paper is organized as follows. In section 2 we give a very brief introduction to the internals of GCC to give the reader a basic understanding of the encountered problems with the port. Section 3 highlights some design decisions taken, mostly dictated by the C6x microarchitecture. In section 4 we provide a very basic C source code example and the output generated by our compiler. The output is discussed and critically evaluated. The paper finishes with a short summary and description of the next project stages in section 5.

2 The GNU Compiler Collection (GCC)

The GNU Compiler Collection (GCC) is probably one of the most popular open source software project. It is distributed under the GNU General Public License, which grants free usage but requires distribution of the source code along with every binary of a derived project. GCC is available for a large number of processor architectures, among them IA-32, IA-64, ARM and PowerPC. On the other hand, DSP support is very scarce: to our knowledge, only the TMS320C4x target belongs to the official GCC distribution. There exist unofficial patches for the A21xxx (Sharc) and 21xx DSPs by Analog

Devices as well as for Motorola's 56k processors but the current status of these projects is unclear. Their development is not connected with the evolution of GCC.

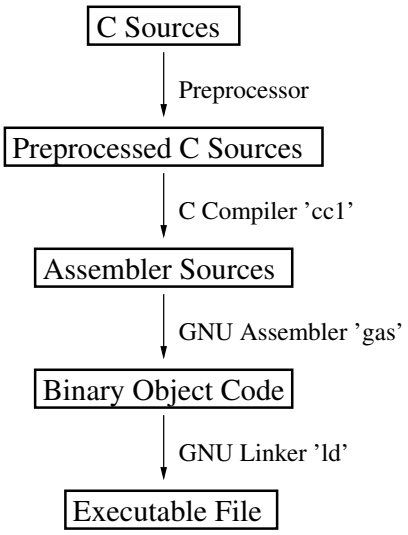

Figure 1: GCC Tool Chain

GCC consists of a pipeline of tools, which it uses to process the high-level language sources all the way down to the final executable (figure 1).

Additionally, there are a number of further tools, such as the disassembler *objdump*, the Binary Format Descriptor (BFD) library, which is responsible for generating the correct output format, or the GNU Debugger GDB. The compiler alone sums up to roughly 400 000 lines of code.

The compiler itself consists of three phases (table 1). This modular structure greatly simplifies extending GCC. If a new programming language is to be supported, another front-end must be written. Adding a new target architecture means developing a new back end, as is the case in our project. The middle-end performs most of the optimizations by manipulating the internal representation of the program to be translated, the so-called Register Transfer Language (RTL).

The implementation of a new backend consists primarily of implementing each a set of target macros, expander definitions and instruction patterns. Target macros are responsible for

Phase	*Purpose*
Front End	Translation HLL → RTL
Middle End	Optimization Passes
Back End	Translation RTL → Assembler

Table 1: Compiler Stages

technical aspects, such as definition of stack frame layout, calling conventions, purpose of registers or width of operands. Expander definitions are used to transform the tree representation of C functions into RTL expressions which reflect the capabilities of the target architecture. In a later stage, a matching process uses instruction patterns to transform RTL expressions into one or several assembler instructions. Figure 2 gives an example for an instruction pattern defining an add operation.

```
(define_insn "addsi3"
    [(set <Template-Variable 0>
        (plus:SI
            <Template-Variable 1>
            <Template-Variable 2>))
    ]
    ...
    "add %1, %2, %0")
```

Figure 2: Instruction Pattern Example

Suppose it is matched by the RTL expression

```
(set (reg 5) (plus (reg 7) (reg 8)))
```

, it could emit a corresponding assembler statement `add A7, A8, A5`.

During the course of this project, approximately 140 target macros, 26 expander definitions and 21 instruction patterns have been implemented, although by far the majority of project time was consumed by analyzing GCC source code and understanding its structural and operational aspects.

3 Design Aspects

As many DSPs do, the C6x architecture lacks hardware support customarily encountered in contemporary general-purpose processors. This

certainly complicates porting GCC. Surely, GCC was developed with other microarchitectures in mind.

Firstly, C6x does not support the notion of a stack. There are no dedicated stack pointer and frame pointer registers as well as corresponding `push` and `pop` operations. Therefore, some of the general purpose registers have to be sacrificed; we chose B15 as stack pointer and A15 as frame pointer.

The layout of a stack frame is depicted in figure 3; it is chosen somewhat arbitrarily. The picture illustrates why both stack and frame pointer are necessary: the length of the "Dynamic Variables" block is variable at runtime (cf. the `alloca()` machine-dependent C library function).

Secondly, the C6x architecture does not provide a call-return mechanism. The only means of leaving sequential execution are a branch instruction or an interrupt. Having implemented a stack as described above allows us to emulate the call-return. Additionally, there are no `enter` and `leave` instructions as in Intel's IA-32 architecture to ease the management of individual stack frames. Instead, the corresponding functionality must be implemented "by hand".

Furthermore, C6x neither provides a multi-bit condition code register nor a universal compare instruction. Instead, it provides specialized compare instructions as `CMPEQ` and `CMPGT` and only a single bit for the representation of truth values. This must be reflected in code generation for conditional expressions.

4 Results

So far our preliminary port supports all C control constructs, such as *for*, *while*, *switch-case* etc. Macros for many operators are still missing, but implementing them should be fairly straightforward.

Figure 4 depicts a very short C fragment. It demonstrates variable definition, assignment, a comparison and an arithmetic shift operation.

The corresponding output of our compiler is shown in figure 5.

Let's try to interpret the output. The first four instructions after the `main` label constitute the function prologue. The "old" frame pointer

```
int main(void)
{
  int a = 3;

  if (a<34)
    a >>= 4;

  return 12;
}
```

Figure 4: Simple C Example

```
        .text
        .align 32
        .global  main
main:
        add b15, −4, b15
        stw a15, *b15
        mv b15, a15
        add b15, −4, b15
        mvkl 3, a0
        stw a0, *−a15(4)                    10
        ldw *−a15(4), b1
        nop 4
        mvkl 33, b2
        cmpgt .L2 b1, b2, b0
        [b0] mvkl .S2 L2, b1
        [b0] mvkh .S2 L2, b1
        [b0] b .S2 b1
        nop 5
        ldw *−a15(4), b2
        nop 4                               20
        shr b2, 4, b2
        stw b2, *−a15(4)
L2:
        mvkl 12, a0
        mv a0, a4
        add a15, −4, b15
        mv a15, b15
        ldw *b15, a15
        nop 4
        add b15, 4, b15                     30
        ldw *b15, a14
        nop 4
        add b15, 4, b15
        b .S2 a14
        nop 5
```

Figure 5: Generated Assembler Code

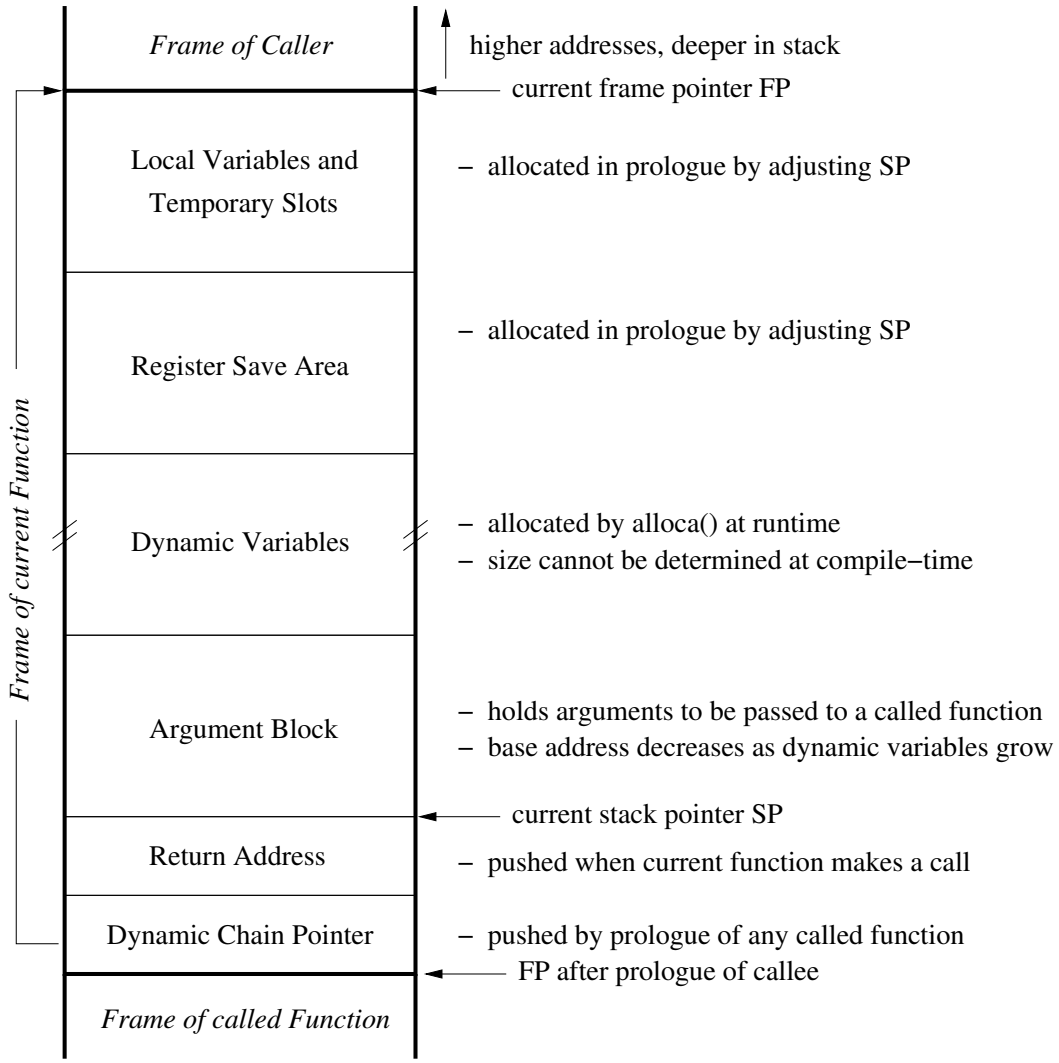

Figure 3: Stack Frame Layout

is saved onto the stack, the new FP gets the current value of the SP and space for the local variable a is allocated on the stack.

After the prologue, the body of `main` starts. Variable a obtains the value three. Note that the compiler generates `if (!(a>33))` instead of `if (a<34)` due to the unavailability of a "lesser-than" instruction pattern. The conditional jump is implemented using the unique *conditional execution* feature of the C6x architecture. If the branch to label `L2` is not taken, variable a is loaded, right-shifted four bits and written back to memory.

The function epilogue starts at label `L2`. The return value is transferred to register a4. The finishing clean-up consists of

- deletion of the current stack frame,
- restoration of the previously saved FP,
- load of the return address into a14 and
- an register-indirect branch ("jump") to that address.

From the generated code it can be observed that there is no machine-dependent optimization. WhereasWhile the compiler middle-end

is responsible for machine-dependent optimizations such as loop-hole detection, dead code elimination etc., our backend does no optimization at all. Instructions are not packed into VLIW words yet, and necessary delay slots are incorporated in the form of NOP instructions (consider for example the branch instruction at the end).

5 Conclusions & Outlook

We have demonstrated that it is possible even with very limited resources to craft a DSP compiler on the basis of the GNU Compiler Collection. We hope to close the remaining gaps and to convert the somewhat prototypic compiler into a powerful development tool as it is already the case for many other processor architectures.

Among the most important problems to be solved are the following:

- The full set of arithmetic operations must be supported. For example, our compiler lacks target macros for logic operators.
- Machine-dependent optimization is indispensable.
- Floating-point data must be integrated (and a way of supporting the C67 DSP must be found).
- A suite of C conformance tests is necessary.
- A maximum of compatibility to the code generation tools by TI is desirable.
- The tool chain must be completed.

The two most needed features in code optimization are delay slot and instruction scheduling and VLIW instruction word packing, as our code example demonstrated. Current work at the Real-Time Group at Chemnitz University is focused on optimization issues.

The conventions for function calls (which arguments to pass in registers etc.) and the layout of types in memory should be made compliant with those described in Chapter 8 of [5]. This might open the way for linking GCC-compiled object code against object code compiled by Texas Instruments' C6x Code Composer Studio.

As far as the GCC tool chain is concerned, we are making progress. At the time of writing this paper, the port of the GNU assembler is almost done, we have a functioning BFD library producing COFF output and the disassembler *objdump* is up and running. The latter is even able to analyze output of the TI development tools.

We invite interested people to the project's web pages at

http://rtg.informatik.tu-chemnitz.de

where we offer documents describing in detail the compiler port ([3]) and other aspects of the tool chain ([4]), as well as current CVS snapshots of the tools. The software is distributed under the GNU General Public License (GPL).

References

[1] Robert Baumgartl, Ingo Oeser, Daniel Schreiber, Michael Schwindt: *DSP Accelerator Support for Linux*. Proceedings of ICSPAT'00, Dallas, 2000

[2] E. Bianchi, L. Dozio, G. L. Ghiringhelli, P. Mantegazza: *Complex Control Systems, Applications of DIAPM-RTAI at DIAPM*, Realtime Linux Workshop, Vienna, 1999

[3] Jan Parthey: *Porting the GCC Backend to a VLIW Architecture*. Diploma Thesis, Chemnitz University of Technology, March 2004

[4] Adrian Strätling: *Extending the GNU Assembler for Texas Instruments TMS320C6x-DSP*. Term Paper, Chemnitz University of Technology, March 2004

[5] Texas Instruments, Inc.: *TMS320C6000 Optimizing Compiler User's Guide*. October 2002

[6] uCLinux Embedded Linux/Microcontroller Project. *http://www.uclinux.org/*