

## **TASKING helps Siemens with 32-bit TriCore™ architecture design**

Published: Embedded Systems Programming Europe

Date: November 1997

*By Peter Hoogenboom  
TASKING Software BV  
Amersfoort, The Netherlands*

---

### **Introduction**

Historically the standard procedure for developing a tool set (e.g. C/C++ compiler package) for a particular processor was straightforward. The chip manufacturer supplied the architecture specification to one or more external software companies and wished them good luck. By that time the architecture was frozen and chips were being produced, either in samples or mass volume, feedback from the software companies could no longer be incorporated in the chip design. However, sometimes feedback was used in the next proliferation of the core. The Siemens C167 16-bit microcontroller is a good example of the latter. The C167 has some additional instructions to the C166. Some of these instructions were requested by TASKING as they are very useful in helping C compilers generate improved code for accessing data anywhere in the 16MB address space, yet keep the 16K paged approach. It was probably with this experience in mind that Siemens changed its attitude towards chip design. So, when the TriCore project was started, Siemens decided that software needs had to be considered during the design phase of the architecture and invited a compiler expert from TASKING to join the Siemens architecture design team in San Jose. This article describes the outcome from a software perspective of the TriCore design process.

### **Requirements**

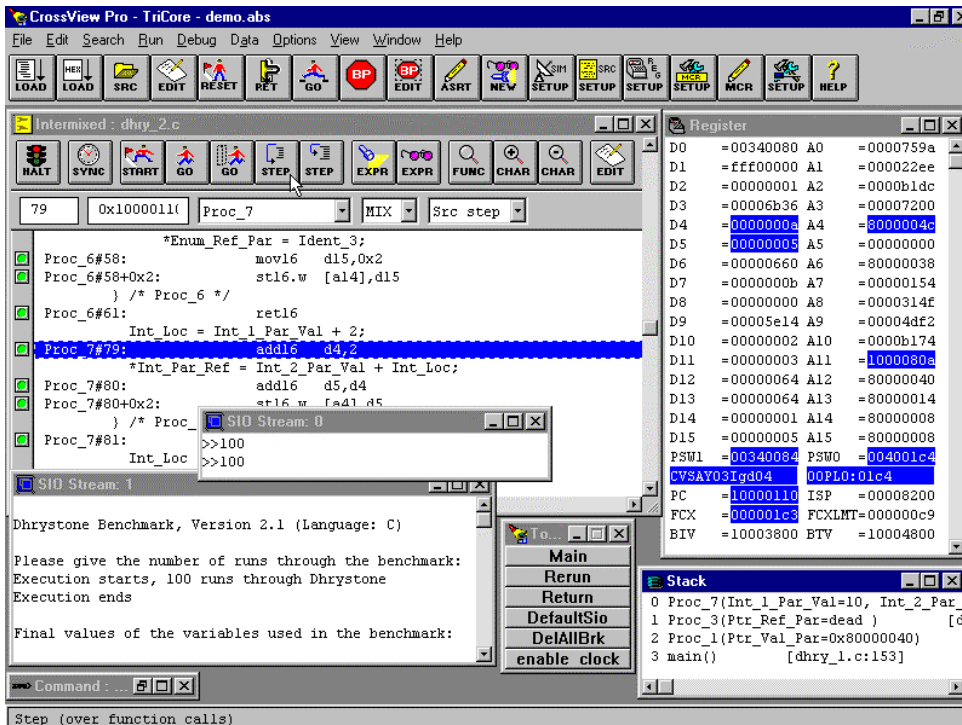
The 32-bit TriCore architecture is designed for use in embedded systems and combines microprocessor, microcontroller and DSP technology in a single core. Each technology area has its own hardware and software requirements. The typical microprocessor area deals with RISC principles, pipelines, super-scalar execution and code/data caches, whereas the microcontroller part supports on-chip peripherals and memory, real-time (deterministic) behaviour and fast context switching. The DSP implementation of the TriCore is also very complete. It's more than just a multiply-accumulate unit, but includes instructions for fixed-point arithmetic and saturation. It is obvious that all these features must be supported by the software tools, especially the C compiler. However, besides these three major building blocks there is one very important generic issue for the entire chip: it must be used in embedded systems! This results in two additional requirements: cost of silicon and code size (read cost of memory). There was also some good news! The good news was that Siemens did not have any legacy code in the 32-bit area, so the entire project was a clean room design. Isn't this the dream of every hardware and software engineer?

## Getting the project started

Well, such an extensive project is not a trivial task. For instance, how do you know that you meet any of the requirements mentioned above beforehand? The answer is relatively simple: measure TriCore performance and compare it to existing processors. So, as well as analysing code generated by compilers for competitive processors, the following needs for gathering TriCore data were defined:

- A model of the TriCore architecture
- Rapid prototyping software tools
- A reliable and robust interface between the software tools and the model

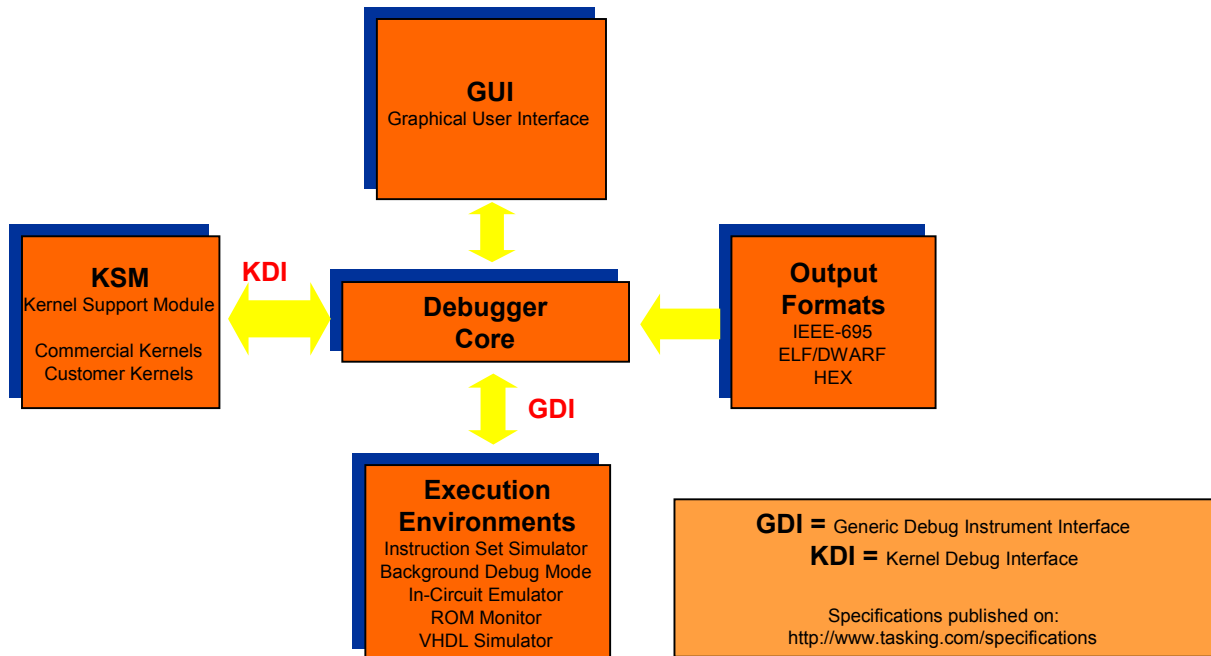
Based on the initial design, Siemens created a plain instruction set simulator engine, modelling the behaviour of the architecture. In parallel with this, TASKING created a rapid prototyping C compiler, absolute assembler and CrossView Pro source level debugger. The prototype C compiler was of course a limited implementation (restricted optimisation and target independent floating-point support). However, it proved to be adequate for benchmarking. The rapid prototyping tools were used to inspect the effect of proposed instruction set changes. Since a significant number of architectural changes were expected during the design phase, a central instruction set database was created. This database could be modified easily and was also used to generate information for the instruction set simulator. To eliminate human mistakes and avoid inconsistencies between the model and the tools, TASKING built a converter to generate tables, which were used by the assembler, debugger and compiler, from the same instruction set database. This proved to be a very powerful technique, because often an assumed assembler error turned out to be a problem in the instruction set database.



## Hooking up the CrossView Pro debugger

TASKING has developed two important open standards to guarantee easy interfacing and access to the CrossView Pro debugger by kernel and emulator manufacturers: KDI (Kernel Debug Interface) and GDI (Generic Debug Instrument Interface). The connection between CrossView Pro and the execution environment (e.g. simulator, ROM monitor, ICE) has been defined in GDI. The GDI standard is adopted by an increasing number of emulator and debugger vendors as it contains a well-documented and extendible API, suitable for implementation as a Windows dynamic link library (DLL). CrossView Pro for TriCore is connected to the Siemens simulator engine (TSIM.DLL) using

GDI. With the rapid prototype tools and CrossView Pro in place, we are now ready to judge benchmark results, answer architecture related questions and make the correct design decisions.



## What should the register file look like?

One of the first design decisions to make was related to the register file. The initial design of the TriCore included a unified register file consisting of 32 registers of 32 bits. With 32 equivalent registers, you require 5 bits to encode a register number in the instruction. One of the major design goals of the TriCore is to have a high code density, and the most effective way to achieve that is the introduction of 16 bit "short" instructions in addition to the normal 32 bit instructions. The gain is optimal when the most frequently used instructions are encoded in 16 bit. Most 16 bit instructions need at least 2 registers, which requires 10 bits to encode the register-pair. To simplify instruction decoding, a fixed bit in the instruction is needed to distinguish between 16 bit and 32 bit instructions. So, from the 16 bit instructions already 11 bits are used for register encoding! This leaves only 5 bits to make 32 different instructions. The question is whether that is enough or not. We used a C166 and MIPS R3000 C compiler to investigate the instruction distribution of compiler generated source code. Analysing the code for a number of different benchmark programs showed that the typical instruction distribution is such that a set of only 32 "short" instructions is not optimal in terms of code density. So, this scheme would not meet the code size requirements of embedded systems.

Dividing the register file into 16 data registers and 16 address registers means that only 4 bits are needed to encode a register in an instruction. Since most instructions either operate on data registers only or address registers only, you can now encode 127 instructions with two register operands in 16 bits. One concern about a split register file approach is that it could result in additional instructions to move the contents of a data register to an address register and visa versa. Analysis showed that the need for these additional move instructions is limited. Normally, integers (data register) and pointers (address register) are not mixed in C/C++ expressions. Exceptions that should be addressed in the architecture are:

1. **Pointer +/- scalar.** Used for array subscripts, and accessing a structure member via a pointer. A C compiler frequently uses these calculations, so dedicated instructions to add or subtract a scalar to or from an address register were devised. The instructions even allow a scaling factor of 1, 2, 4 or 8 to be specified, since in C an array index must be multiplied by the size of the base type, which is often a 1, 2, 4 or 8 byte quantity.

2. **Subtract two pointers.** Used to calculate the number of elements between two pointers into the same array. A dedicated instruction was added for this. Subtracting two pointers in C automatically implies a division by the size of the base type, so a scaling factor of 1, 2, 4 or 8 may be specified here also.
3. **Conversion from scalar to pointer and vice versa.** These conversions are not performed automatically by the C compiler, but must be requested explicitly by using a *cast*. These types of conversions are not normally used in well written C code. MOV instructions were added to move the contents of a data register to an address register and vice versa. Although these instructions will not be used very often, they cannot be omitted.

## Little and big endian

Another major decision was whether to use "big endian" byte ordering or "little endian" byte ordering. From a technical perspective, both approaches are acceptable, as long as it is used consistently throughout the whole architecture. As the Siemens C166 is a little endian architecture (the only compatibility issue left), it seemed logical to choose little endian for the TriCore architecture as well. However, there is also a need to support peripherals that are big endian. Solving this issue in software would result in a C language extension (which always should be kept to a minimum) for big endian memory mapped I/O ports and run time overhead swapping bytes. Both were not acceptable. For that reason, the bus interface contains logic to swap the endiannes depending on the device being accessed.

## 16 bit instruction selection

The initial selection of "short" 16 bit instructions was done by hand. We refined the instruction selection with some statistical data gathered by post-processing the generated output of the C166 and MIPS R3000 C compilers for a number of benchmark programs. When the first version of the prototype C compiler became available, it was possible to get statistical data for the TriCore instruction set itself, so that a better refinement was possible. Initially, there were short load instructions to "push" and "pop" an object onto or from the stack:

```
ST16.W [-SP]4,D0
```

```
LD16.W D0,[SP+]4
```

Frequently, it is more efficient and simpler to use a fixed size stack frame and access the stack variables with the indirect-with-offset addressing mode so these instructions are never generated by the C compiler. As a result, these short instructions were abandoned and the opcodes were used to add other, more useful short instructions. Examples of short instructions that were added as a result of an analysis of the code, generated by the prototype C compiler, are several conditional branch instructions with limited constant and/or offset fields, and indirect load/store instructions with auto-increment.

## Pipeline optimisations

The architecture supports two execution units: one for arithmetic on the data registers and one for address arithmetic. In general, an arithmetic instruction operating on data registers and a load/store instruction can be executed within the same cycle. An instruction scheduler in hardware would increase the cost of silicon too much, so pipeline optimisation is now done in software. To utilise the parallel execution mechanism, the assembler rearranges instructions, so the pipeline is filled in the best possible way.

## Instruction set changes

Apart from the changes in the selection of short (16 bit) instructions, some other changes to the instruction set were influenced by the design of the C compiler. For instance, there was no instruction to subtract two address registers and deliver the result in a data register in the initial instruction set. This instruction is useful to implement a C pointer subtraction. As a result, the instruction DIFSC was

renamed to SUBSC for consistency, and a new DIFSC to subtract two address-registers, was introduced. A specific short instruction was added to subtract a constant from the stack pointer. This instruction is used by the C compiler in the function prolog code.

## Calling convention

An important issue in the design of a C compiler is the definition of the calling convention. The calling convention has a major impact on the performance of a C application. The optimal calling convention strongly depends on the architecture and instruction set. During the development of the instruction set, several different proposals for a calling convention were discussed. At some point, we had the need for statistical data on the typical number of register variables and stack frame size. To get this data, a MIPS R3000 C compiler was modified to output the required information for every compiled function.

For a processor with as many registers as the TriCore, it is advantageous to pass function parameters in registers. A fixed set of registers is dedicated to parameter passing. This set must be large enough to hold all function parameter for the majority of the function calls. Of course, the design of the calling convention was closely related to the discussions about the split versus unified register file. For both types of register file, a calling convention and sample function prolog and epilog code was written and discussed in the architecture team. Different hardware implementations of the call and return instructions were devised and their implications on the calling convention were discussed. Initially, the call and return instructions used the stack to save the return address. In this implementation, registers that must be preserved across a function call should be saved on the stack in the function prolog and restored in the function epilog code. Later, new call and return instructions were defined, that take advantage of the wide bus to on-chip RAM memory. On a CALL, 16 registers are automatically stored in memory and added to a linked list of "context blocks". This change in the instruction set drastically improved the execution speed and code size for function prolog and epilog code!

