An Embedded System Case Study: the FirmWare Development Environment for a Multimedia Audio Processor

Clifford Liem[1,3], Marco Cornero[1], Miguel Santana[1], Pierre Paulin[1], Ahmed Jerraya[3],
Jean-Marc Gentit[2], Jean Lopez[2], Xavier Figari[2], Laurent Bergher[2]

1. Central R&D, SGS-Thomson Microelectronics, 850, rue Jean Monnet, BP 16, 38921 Crolles, France
2. Thomson Consumer Electronic Components, 5 bis, chemin de la Dhuy, 38240 Meylan, France
3. Laboratoire TIMA, L'Insitut National Polytechnique de Grenoble, 46, ave Félix Viallet, 38031 Grenoble, France

Abstract

This paper outlines a case study at SGS-Thomson Microelec-tronics on the development of a firmware development environment in co-operation with Thomson Consumer Electronics Components. The envionment is for an embedded processor used for audio decompression algorithms including: MPEG2, Dolby AC-3 Surround, and Dolby Pro-logic. The enabling component of the firmware environment is a retargetable compiler which maps high-level algorithms onto the embedded processor. Although compilation is the critical technology, this experience has shown that it is insufficient and that other supporting design tools are also important. For this project, that environment includes an instruction-set simulator, a source-level debugger, a custom linker, and a compiler validation strategy. The methodologies are outlined in this paper with an emphasis on the lessons learned in this hardware-software team development.

1 Introduction

Keeping abreast of the evolution of the MPEG standards set forth by the International Standard Organization (ISO) implies incorporating a solution which balances high performance with flexibility. An instruction-set processor can meet these flexibility demands through embedded firmware, while customizing the processor for the right application characteristics ensures the required performance.

The embedded system presented here targets the MPEG2, Dolby AC-3, and Dolby Pro-logic audio decoding standards. The product areas of interest are: DVD (Digital Video Disk), multimedia PC, set top boxes (satellite), High Definition Television (HDTV) and high-end audio equipment. The key component of the system is a custom instruction-set core with special features for the performance requirements of this arena. In addition to being a stand-alone product (the STi4600 [1]), the architecture can also be embedded as a core for integrated products.

The hardware and software environments required close communication between two teams. The hardware team at Thomson Consumer Electronic Components developed a high-performance processor core and instruction-set model whose features are a result of a thorough analysis of the time-critical functions of the MPEG2 and Dolby AC-3 standards. The software team at SGS-Thomson developed an optimizing compiler, source-level debugging interface to the instruction-set model, a custom linker, and a validation environment for the development of application code on the platform system. The instruction-set of the embedded processor served as the rigid contract between the development of the hardware and the software tools. Throughout this process, exchanges between the two teams served to refine the architecture and solidify the firmware development environment.

Permission to make digital/hard copy of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage, the copyright notice, the title of the publication and its date appear, and notice is given that copying is by permission of ACM, Inc. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and /or a fee.
DAC 97, Anaheim, California
(c) 1997 ACM 0-89791-920-3/97/06 ..$3.50

This paper presents details of the development experience with an emphasis on the lessons learned in the co-operation of a hardware and software team. The rest of the text is organized as follows: Section 2 describes the background of the project followed by a description of the processor architecture in Section 3. Section 4 reviews the principles of the retargetable compiler environment. Section 5 describes the architecture specific tools which were developed to complete the firmware development environment. Section 6 describes the compiler validation strategy and results, followed by a conclusion in Section 7.

2 Project Background

This project begins with a history of using a well-defined design process for an instruction-set architecture [2]. This methodology includes the use of a special macro-assembler known as *RTL-C*. As described in [2], source code is written in a form which follows the syntax of C with a number of strict guidelines. Variable names refer to specific registers of the architecture and operators map directly onto operations in the architecture. For operators that do not exist in C, built-in functions are used. Each line of C refers to parallel executing operations on the processor.

While this style is restrictive for high-level coding, it has strengths over pure assembly coding specifically in the initial architecture definition phase. The use of the C language syntax allows a second path of compilation on the workstation for functional validation. Moreover, it allows the use of standard Unix profiling tools to measure real-time performance; for example, the profiling functions of the cc and gcc compilers, and the profiling utilities: tcov, prof, and gprof.

However, when the hardware is stable, software development productivity is low since application code must be written on a level comparable to macro-assembly. With the increasing complexity of the MPEG audio standards, the need for a full C compiler had arisen. The requirement for the compiler was that it allow higher productivity by permitting code to be written on a more abstract level and that it not compromise the quality (performance and size) of any code that can be written at the assembly level.

The tuning of an embedded processor system to meet the demands of an application area is a common practice [3][4][5]. Reuse is key to this design style and some have even put in place the methodologies to support this flow (e.g. EPICS [6]). Unfortunately, the firmware tools which can be easily adapted to evolving instruction-set architectures are few. This is a direct consequence of today's poor state of firmware development tools, especially compilers for DSPs [7]. Work addressing the needs of the embedded instruction-set processor methodology is just beginning to appear [8][9].

3 Architecture Description

The MMDSP architecture is a Harvard, load/store instruction-set processor based on VLIW concepts. Communication is

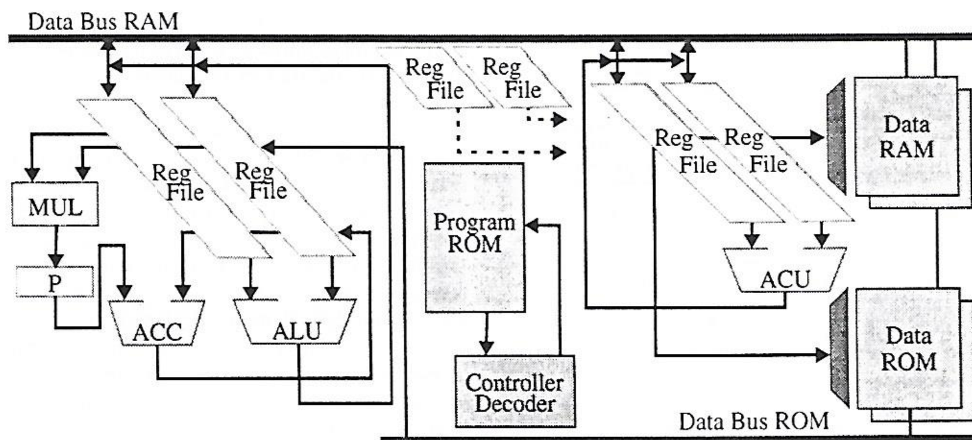

Figure 1. Thomson Consumer Electronic Components MMDSP Architecture Block Diagram

centralized through a bus between the major functional units of the ALU (Arithmetic and Logical Unit), ACU (Address Calculation Unit), and memories. The program control unit is a standard pipelined decoder with the common branching capabilities (jump direct/indirect, call/return), but also including interrupt capability (goto/return-from interrupt) and hardware do-loop capability. Three sets of registers are used to provide three nesting levels of hardware loops which can be increased without limit by pushing any of these registers onto the run-time stack.

Although the basic design of the unit can be compared to many classic DSP architectures, there are certain features which set it apart and allow it to perform well in this application domain. The post-modify ACU includes custom register connections and increment/decrement capabilities which allow it to efficiently traverse the special memory structures. This includes increment and decrement by selected constant values, as well as increment values held in dedicated registers. One last possibility is the capability to use a constant increment value coming from the instruction word. With this large number of possible operations that can be performed on the ACU, certain combinations are chosen to be encoded in the instruction set such that they execute in parallel to other operations.

The ACU has been designed to work in concert with the memory organization which has been developed around the data-types needed for the MPEG audio algorithms. A first partition separates memory into ROM mostly for constant filter coefficients, and RAM to hold intermediate data. For each of these memories, several data types are available, some are high precision for DSP routines, others are lower precision mainly for control tasks. In addition to the standard memory locations, there are memory-mapped I/O addresses for communication with the peripherals.

The MAC (multiply-accumulate) unit was designed around the time-critical inner-loop functions of the application. The unit has special register connections which allow it to work efficiently with memory-bus transfers. In addition, certain registers may be coupled to perform double precision arithmetic.

4 Retargetable Compiler Development

The compilation system was developed using the rule-driven approach of the FlexWare project [9][10]. For clarity, this section reviews the main principles of this approach, while the following section will give an outline of the architecture-specific compiler developments for this project.

4.1 Rule-driven compilation

This compilation approach is based on step-wise progressive refinement presented by Gurd [11]. This programming environment for compiler development uses a process which is divided into four main phases shown by example in Figure 2.

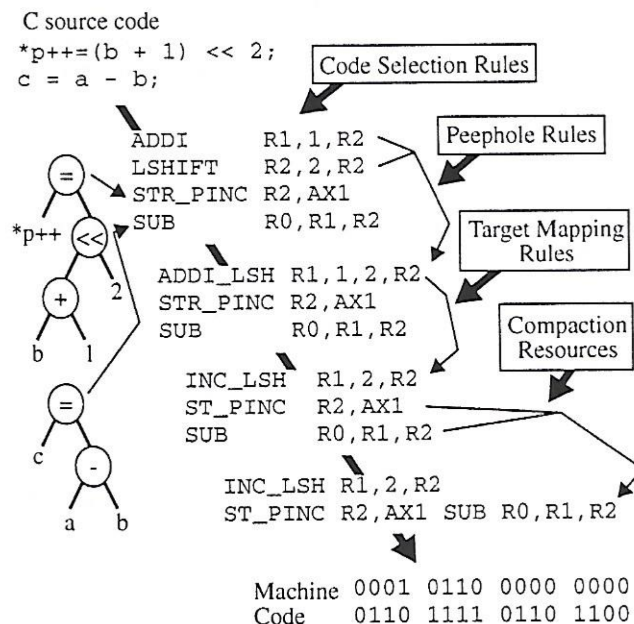

Figure 2. Rule-driven Compilation Example

The approach builds on many traditional compiler techniques allowing an open-programming concept to be applied as rules at each step. The steps are as follows:

1. Sequential code selection. The developer defines a virtual machine which resembles in functionality the instruction-set of the real machine, but is sequential in operation. Parallel execution streams are simplified to one stream. The virtual machine description contains two main parts: 1) a description of resources including register sets and addressing modes, and 2) a set of code selection rules.

For the code selection rules, the developer defines the mapping between the C code onto the virtual machine instruction-set. For each operation which may occur, the developer provides a rule which is defined in a programming language. This rule will be triggered upon matches to the source code and executed at compile time. This approach allows the developer to provide simple rules for the most common cases and more complex mappings for special features of the architecture. For example, the developer may restrict the use of certain registers whose function are constrained by the architecture. This is important to support special-purpose registers which are often found in em-

bedded processors. Register assignment within register sets is performed after code selection using a coloring approach. This is done in a manner which satisfies the constraints imposed by the selection rules.

2. Optimizations. Instructions for the virtual machine may be passed to a series of optimization routines, such as a peephole optimizer which transforms sequential occurrences of operations into more efficient operations through simple replacements. As the transformations are activated recursively, the code may be significantly improved. At this point in the compilation, it is also possible to add custom optimization sequences, since the input is very well defined (more in Section 5). This was done in the definition of the virtual machine.

3. Mapping to the target machine. The optimized sequence of virtual instructions are transformed into operations for the real machine. Each transformation again follows a rule provided by the developer. Each rule indicates a source piece of code and a target implementation in the form of micro-operations representing bit fields of the instruction-set. A rule is defined in a well-structured programming language.

4. Code compaction. Micro-operations are compacted into instructions. At this point the full parallelism of the machine is recuperated. The compaction procedure follows rules of both bitfield formats and read/write/occupy resources which are indicated by the developer. The compactor attempts to push the maximum number of micro-operations to the earliest possible positions. The straight-forward tasks of assembly and linking follow compaction.

Although the rules for this type of compiler must be written by an experienced developer, the retargeting time is relatively short. The main strength of rule-driven compilation is the inherent flexibility of the approach. The compiler developer has the means for describing specific rules and strategies for efficiently mapping higher-level constructs onto the processor, based on his knowledge of the architecture idiosyncracies. It is also a means to leverage previous compiler development experience.

When compared to a traditional compiler approach, the rule-driven approach allows for faster development, and is easier to retarget. The quality of the results depend on the compiler development and optimization effort.

When compared to model-based retargetable compilation approaches [12][10], the rule-driven approach requires retargeting development time; whereas, in principle, a model-based compiler requires only a small change to the model to arrive at a new compiler. Also, the rich data structures used in those systems potentially allow for more sophisticated optimizations. However, in our experience [9], this is compensated by the applicability of the rule-driven approach to a very broad set of processor architectures, from low-end microcontrollers to VLIW DSPs and the ability to handle architecture idiosyncracies on case-by-case development strategies.

4.2 Coding Style

This compiler environment allows C source coding on various levels of abstraction which may be mixed. In addition to the ANSI C behavioral level of coding, lower levels of abstraction are provided. This is to allow the designer to reach all the functionality of his processor, at perhaps the expense of code portability. Our experience has shown that while the effort on developing optimizations is important to achieve a more portable level of code [13], the capability of a compiler to handle lower levels of coding is essential. Especially in this time when retargetable compilation techniques are immature, it is important that the bridge to higher levels of automation be crossed as smoothly as possible.

We define four coding levels summarized as follows, starting

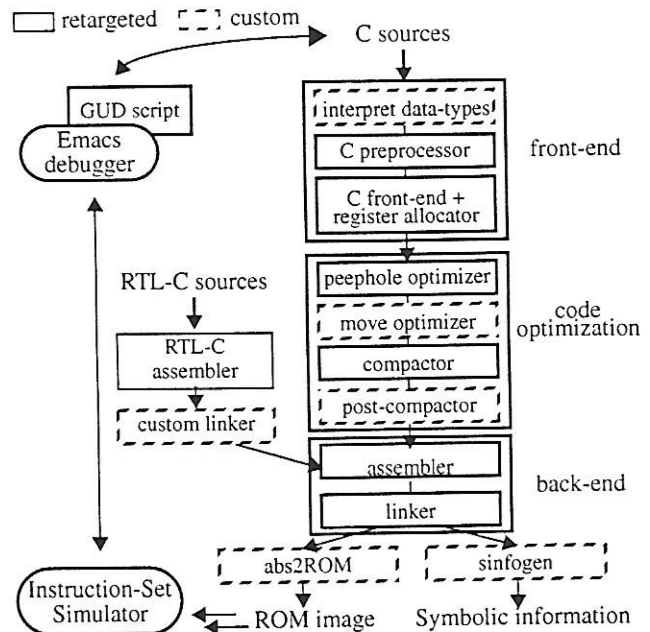

Figure 3. Full Compiler Suite and Development Environment

with the most abstract level:

1. High Level:
   Behavioral ANSI C. This level is characterized by the use of variable, structure and array references and all operators in C.
2. Mid Level
   This level allows the use of built-in functions. Any arrays or structures that are declared in memory must be accessed by pointers. Variables and pointers may be user allocated into extended storage classes and register sets.
3. Low Level
   This level allows the user-assignment of variables and pointers in specific registers.
4. Assembly Level
   This level allows the programmer to write in-line assembly directly in C code.

To reemphasize, these coding levels may be mixed. The reason for using lower levels is to reach the full functionality of the chip as directly as possible; however, lower levels (2-4) are used sparingly as they reduce code portability. Level 1 is standard ANSI C. Levels 2 and 3 are arrived upon by small *C-like* syntax extensions which are processor specific. C code which remains within these three levels (1-3) allows a second path of compilation on the workstation, given the provision of the proper masking of C extensions and a bit-true library (Section 5.2.3).

5 Architecture Specific Development

In addition to the retargeting effort of the standard suite of tools, custom optimizations and interfaces were developed to provide a complete compiler environment. Some of these are optimization modules required for higher performance, while others are tools required to interface into the hardware design flow. The complete environment is shown in Figure 3.

5.1 Custom Compiler Development

5.1.1 Custom Data-type Mapping

For this architecture, the first key item to resolve was the support of the custom memory structure. This structure posed a unique challenge which stems from the multiple memories with varying addressing strides.

Inherently, the retargetable code generation system handles

multiple memories and multiple data-types. However, it is required that all memories be of the same bit-width. This implies that data-types of increasing width take either the same or more memory spaces. For example, if there are three data types *dtype1, dtype2, dtype3* where the corresponding bit-widths are such that *dtype1* < *dtype2* < *dtype3*. This implies that if *dtype1* takes 1 memory space and *dtype2* takes 2 memory spaces, then *dtype3* must take 2 or more memory spaces. For this architecture, this is not always true. There exists a larger data-type (*dtype3*) which takes fewer memory spaces than a smaller data-type (*dtype2*). It is designed this way in order to meet the hardware timing requirements.

Many solutions were proposed. The first was to change the hardware. This was not possible because of timing restrictions. The second solution was to extend the memory handling of the compiler. A proposal was written for this solution which was to take several weeks to implement.

Instead of this solution a third one was adopted, which in contrast required only a small change to the compiler and a few days to implement. The solution was to change the compiler interpretation of data-types. The larger data-type (*dtype3*) is recognized as a smaller data-type (*dtype2*) and vice-versa. Therefore, the smaller data-type (*dtype2*) would use more memory spaces than the larger data-type. The solution had only small side-effects. It was required that a small change to the compiler in interpreting constants be made so that constants used with the larger data-type (*dtype3*) were not chopped prematurely. A second concerned automatic variables placed on the stack, which would result in the waste of some memory spaces in some cases. This was solved by some simple coding style rules.

5.1.2 Data-flow Optimization

As shown in Figure 1 the target architecture offers a considerable amount of parallelism. For example, the ALU and the ACU can work in parallel if they do not occupy the data bus at the same time. The instruction format provides orthogonal fields for parallel operations, so that compaction is rather straightforward [14]. However, there are cases where data-flow optimizations must be performed in order to best exploit the available parallelism. The most important one is related to data moves within the ALU registers, which can be implemented either through the ALU, or through the data bus. The best choice is the one that allows another operation to be performed in parallel with the move (i.e. an ALU operation if the move is performed through the data bus, or any other operation occupying the data bus if the move is performed through the ALU). A custom *move optimizer* was implemented to improve the results of classic compaction. The utility keeps track of the operations that can be implemented in parallel with moves keeping track of the data-dependencies. It then selects the best move operation by evenly distributing the resource occupation, maximizing the potential parallelism, which is physically done later in the compaction phase.

5.1.3 Post Compaction

Although the processor has a 61 bit instruction word, the high amount of parallelism means that not all processor operations can be encoded orthogonally. This means that certain sets of operations are chosen for parallel operation, while others must be sequential. In addition, the designer has chosen to implicitly encode operations into the instruction word by imposing restrictions on register usage, functional units, data-paths, etc.

This processor has a few encoding schemes which are beyond the capabilities of traditional compaction. To enhance the capabilities of the compactor, we added a post-compaction phase which immediately follows compaction. This post compaction phase is built upon the peephole optimization approach. Rules are provided which search for sequences of assembly operations and replace these sequences with compacted sequences. Resources may be defined as part of a rule so that no data-dependencies are violated during compaction. We applied the post compactor to the encoding restrictions specific to this instruction-set. It performs well for the regions of code where optimization is possible; however, the classic *phase coupling* problem with previous compilation tasks remained (e.g. register assignment is determined before compaction). While this cannot be avoided, the problem occurs rarely.

5.2 Interfacing into the Design Environment

With the hardware design environment well established, the integration of the firmware development environment required the development of certain interfaces. For validation purposes, the hardware design team developed an instruction-set model. Interfaces were implemented to this model from the compiler and to a debugging interface. Furthermore, a bit-true library was developed for the compilation path to the workstation.

5.2.1 ROM Generation and Custom Linker

Customizing the bit format for both the program ROM and data ROM was done to interface to the hardware environment. This was a straight-forward task of format conversion and was anticipated from the beginning of the project.

What was not anticipated was the development of a custom linker to integrate the macro assembly code (RTL-C) with the C application code. As explained in Section 2, the refinement and definition of the hardware was done by writing time-critical portions of the code on a low level (RTL-C). This historical code investment is tapped only by integrating it with the application code or rewriting it in C.

The linking strategy which was developed is shown in Figure 3. The binary code produced by the RTL-C compiler is treated as absolute data in a specific location. The assembler passes this block along with the code produced by the retargetable compiler to the linker. Since the code produced by the retargetable compiler is relocatable, the linker is able to find an absolute position other than the position of the RTL-C code.

5.2.2 Source-Level Debugging

The instruction-set bit-true model, developed to simulate the processor architecture, was implemented with a standard interpretative interface. It contains a set of interactive textual commands to run simulations, watch registers and memory contents, load data-files, etc.

Although it was not a priority at the departure of the project, a debugging interface for running the instruction-set simulator was desired. However, as the tools were maturing, this interface was re-evaluated as an essential part of the environment. It was the important piece that allowed the verification of the operation of both the compiler and the instruction-set simulator.

The debugging interface was developed as an extension to the popular editor *Emacs*. It runs both under GNU Emacs [15] and XEmacs [16] and is based on the GUD (Grand Unified Debugger) library which also has similar interfaces for other debuggers (e.g. gdb, sdb, dbx, xdb, perldb). The interface is capable of setting breakpoints, cycle-stepping, C line-stepping, watching/printing registers, and printing global variables. The interface includes an automatic retrieval of the appropriate source file with an active line indicator. The system is shown in Figure 4.

To our surprise, only a minimum of source-level debugging information is needed in the symbolic information file to have a usable system (generated by *sinfogen* in Figure 3). The alpha version of the debugger contained only source line correspondance information. This allowed the user to run a simulation by stepping through the source code, which allows at the very least

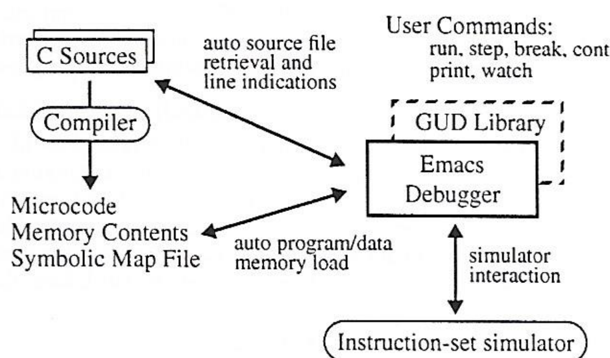

Figure 4. Debugging Interface for the MMDSP Compiler

an assurance of the correctness of the control-flow generated by the compiler. An essential feature is the setting of break points which allows the localization of problems in either the compiler, instruction-set simulator, or bit-true library. This is an extremely tedious task without a source-level debugging interface.

5.2.3 Bit-True Library Development

Our methodology includes a path to compilation on the workstation. For algorithm verification this is an important path to combat lengthy simulation time. However, this is also an essential part of the validation methodology as explained in Section 6.1.

Execution on the workstation must match the bit-true behavior on the processor. This implies the provision of a C library which contains functions for each of the built-in functions defined for the retargetable compiler. For this architecture, the most important of these are the multiply-accumulate functions which have different behaviors depending on the data format being used. In addition, some functions perform automatic rounding and limiting operations.

As many of the data-types of this architecture could be represented using C data-types, the correspondence to workstation bit-true behavior was possible. However, if the types were to diverge greatly, then the bit-true simulation would be more difficult. One solution is to use C++ for the creation of new data-types and overload the operators. We have already had success with this approach in similar projects.

5.3 Lessons in HW/SW Co-development

Throughout the development of this embedded system, a high interaction between the hardware and compiler teams took place. In addition to the high educational value of this concurrent design exercise, one main conclusion can be stated. Each side of the development has its set of complicated constraints. For problems on one side of the coin, the only way to reach a change on the other side is to push a little until the other side either finds a second way within his constraints, or pushes back.

This scenario was indicative in finding a solution to the memory and data-type problem described in Section 5.1.1. From the software side, the simplest solution would have been to change the hardware; however, from the hardware side, the simplest solution would have been to change the software. A formal negotiation following a study of the difficulties on each side was needed to resolve the problem, which by chance fell on the software side.

A similar issue arose which involved the operation of the program stack pointer. The original hardware operation of the pointer caused the inefficient operation of function calls and returns on the compiler side. This would have resulted in slow operation and a waste of either program or data memory. Again, the compiler team saw the easiest solution as the modification of the hardware. In this case, this was a simple change in the hardware and was immediately carried out.

There are no straight-forward answers in the process of concurrent hardware/software co-design. The process is an on-going challenge of staying within the constraints of both sides. The important aspect is a high-level of communication. And, at the very least, the interaction between the hardware and software teams allows solidification of the instruction-set specification, which serves as the rigid contract between the two teams.

6 Experimental Results

6.1 Compiler Validation

The compiler validation strategy that we used is depicted in Figure 5 and proceeds as follows. A test buffer is defined to hold values which are computed by the source code. Data is written to this buffer by means of functions which are contained in two separate libraries: one defined for the workstation compiler, another defined for the target compiler. The source is compiled with each compiler and the appropriate library. The workstation executable is run on the host and the microcode from the retargeted compiler is run on the instruction-set simulator. Following this, the two test buffers are compared. Any differences indicate a problem in the target compiler, and/or the function library, and/or the instruction-set simulator. The workstation compiler is assumed to produce correct code as it has its own validation process.

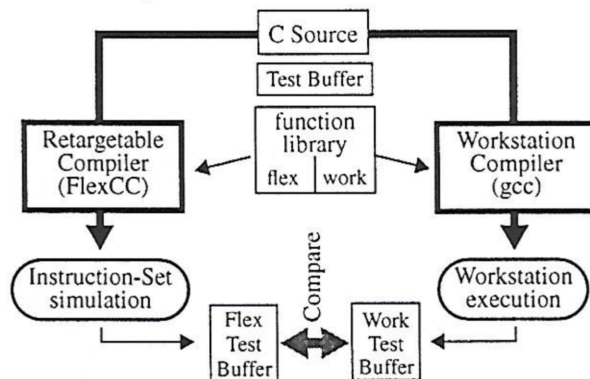

Figure 5. Compiler Validation Strategy

We assembled a set of validation tests in various categories. These are summarized in Table 1. The first categorization breaks up the tests into two large categories: tests which can be used by a broad range of architectures, and others specifically for this architecture. The second categorization separates individual unit tests and full algorithmic tests.

Table 1. Compiler Validation Tests

Type of Test	Category	Operations, Functions	Number of C lines
Generic ANSI C	Unit Tests	bit-op, arith, relation, control, stack, ...	8742
	Integration Tests	bsearch, bubble, btree, gcd, wordcount, malloc, charcount, initptr	2842
Architecture Specific	Low/Medium Level Unit Tests	hardware loops, built-in functions, register sets, special registers	919
	Application Example	FFT	381
		Total	**12884**

Over 12000 lines of C code were run through the validation system, covering all the functionality of the processor that was

expected to be used. This validates a number of tools in the firmware development environment including the retargetable compiler, instruction-set simulator, and bit-true library.

6.2 Compiler Results

In successfully retargeting the compiler to this processor, the requirements set out in Section 2 were met. As explained in Section 4.2, the compiler supports various levels of coding for different types of algorithms. We were able to evaluate the effects of these coding levels on two examples which were manually coded before the availability of the retargetable compiler. The results are shown in Table 2.

Table 2. Code Size Results: Manual vs Compiled

Example	Manual Code Size (RTL-C)	Compiled Code Size (C Compiler)	% Overhead
depack	80	High-Level(1) Source 101	+ 26%
		Mid-Level(2) Source 84	+ 0.5%
		Mid-Low Level(2-3) Source 79	- 0.1%
FFT	235	Mid-Level(2) Source 261	+ 11%
		Mid-Low Level(2-3) Source 228	- 3%

The table shows that for a high-level coding style a code size overhead of 26% is obtained. For a mid-level coding style, a code size overhead between 0.5% and 11% is obtained. The mid-low level coding style meets the code size of the manual code. While level 1. is the ideal level in the interest of clarity and portability, we have found that a mixture with levels 2 and 3 are necessary in time critical portions of the algorithms. For portions of the code which are not time critical, level 1 provides adequate code quality. It is interesting to note that level 4 was not necessary to meet the time constraints, although it is a feature provided by the compiler.

6.3 Recap of Human Effort

Table 3 shows a breakdown of the effort spent on the various activities in the development of the compiler environment. The strong message from this breakdown is that 30% of the effort was spent on validation of the environment. This is an essential part of the design flow. Even if future improvements in compilation techniques reduce the retargeting time, we expect this effort to remain for any specialized embedded processor.

Table 3. Summary of Human Effort

Activity	Effort in Person-Months
Compiler Suite Retargeting	3.5
Custom Compiler Development	1.4
Compiler Validation	2.5
Support / Integration / Porting / Documentation	0.8
Total	**8.2**

7 Conclusion

This paper has presented a case study describing the design of a firmware development environment for a specialized audio embedded processor. The development environment includes a retargetable compiler, an instruction-set model, a source-level debugger, a validation strategy, and interfaces into the hardware environment.

The key lessons learned in this project are as follows:

1. Full environment: Although the compiler is the enabling technology, other tools are important to support the entire design activity. An instruction-set simulator and interfaces into the hardware design environment are critical parts. As well, the value of a source-level debugger cannot be underestimated.

2. Validation: A thorough validation test suite is mandatory, independent of the compiler approach. This constitutes nearly one third of the development effort, which includes the development of a bit-true library. If the application algorithms are available, these are of course the best validation benchmarks.

3. Compiler provision for low-level coding: Our experience shows that a code size overhead of about 30% is common for a high-level coding style. However, our lesson was that the effort put into developing optimizations is a secondary priority after the requirement of the compiler to handle low-level coding styles. The designer must have compiler control so that the timing constraints can be met when the results are not satisfactory.

4. Concurrent design: Hardware and software should be developed concurrently in order to objectively evaluate the constraints on each. Concurrent development between hardware and software teams is always profitable.

5. Optimizations: Techniques which allow higher levels of coding are needed. Although point 3 is the industrial reality, compiler research should continue to find techniques which free the hardware designers from the software constraints.

References

[1] SGS-Thomson Microelectronics, "STi4600 6 channel Dolby AC-3 MPEG 1/2 Audio Decoder Advance Data", January 1997.

[2] L. Bergher, X. Figari, F. Frederiksen, M. Froidevaux, J.M. Gentit, O. Queinnec, "MPEG Audio Decoder for Consumer Applications", *Proc. of the Custom Integrated Circuits Conference*, May 1995, Santa Clara, Ca.

[3] M. Ikeda et al., "A Hardware/Software Concurrent Design for a Real-Time SP@ML MPEG2 Video-Encoder Chip Set", *Proc. of the European Design & Test Conference*, 1996, pp. 320-326.

[4] P. Paulin, M. Cornero, C. Liem, F. Naçabal, C. Donawa, S. Sutarwala, T. May, C. Valderrama, "Trends in Embedded Systems Technology: An Industrial Perspective", in Hardware/Software Codesign, ed. by M. Sami, G. DeMicheli, Kluwer Acad. Pub., 1996.

[5] C. Liem, P. Paulin, A. Jerraya, "ReCode: the Design and Re-design of the Instruction Codes for Embedded Processors", *Proc. of the European Design & Test Conference*, March 1997.

[6] R. Woudsma, et.al., "EPICS, a Flexible Approach to Embedded DSP Cores", *Proc. of the Int. Conf. on Signal Processing Applications and Technology*, Dallas, Oct. 1994.

[7] V. Zivojnovic et al, "DSPstone: A DSP-Oriented Benchmarking Methodology", *Proc. of the Int. Conf. on Signal Processing Applications and Technology*, Dallas, Oct. 1994.

[8] Code Generation for Embedded Processors, ed. by P. Marwedel, G. Goossens, Kluwer Academic Publishers, 1995.

[9] C. Liem, P. Paulin, M. Cornero, A. Jerraya, "Industrial Experience using Rule-driven Retargetable Code Generation for Multimedia Applications", *Int. Symp. on System-Level Synthesis*, Sept. 1995.

[10] P. Paulin, C. Liem, T. May, S. Sutarwala, "FlexWare: A Flexible FirmWare Development Environment", in [8].

[11] R.P. Gurd, "Experience Developing Microcode Using a High-Level Language", *Proc. of the 16th Annual Microprogramming Workshop*, Oct 1983, pp. 179-184.

[12] D. Lanneer, et. al., "Chess: Retargetable code generation for embedded DSP processors", in [8].

[13] C. Liem, P. Paulin, A. Jerraya, "Address Calculation for Retargetable Compilation and Exploration of Instruction-Set Architectures", *Proc. of the Design Automation Conf*, 1996.

[14] A. Lioy, M. Mezzalama, "Automatic Compaction of Microcode", *Microprocessors and Microsystems*, vol 14, no 1. January/February, 1990. pp 21-29.

[15] see site: ftp://ftp.prep.ai.mit.edu

[16] see site: http://www.xemacs.org