Early Creation of Cross Toolkits for Embedded Systems

Nikolay Pakulin and Vladimir Rubanov

Institute for System Programming of the Russian Academy of Sciences, Moscow, Russia

Abstract. Cross toolkits (assembler, linker, debugger, simulator, profiler) play a key role in the development cycle of embedded systems. Early creation of cross toolkits and possibility to quickly adapt them allows using them as early as at the hardware/software codesign stage, which becomes an important success factor for the entire project. Challenging issues for cross toolkits development is efficiency of simulation and CPU instruction set alterations at the design phase. Developing cross toolkits in C/C++ produces highly efficient tools but requires extensive rework to keep up with instruction set changes. Approaches based on automatic toolkit generation from some top level specifications in Architecture Description Languages (ADLs) are less sensitive to this problem but they produce inefficient tools, especially simulators. This paper introduces a new approach to cross toolkits development that combines the flexibility of ADL and efficiency of C/C++ based approaches. This approach was implemented in the MetaDSP framework, which was successfully applied in several industrial projects.

1 Introduction

Nowadays we witness emerging of various embedded systems with rather tough constraints (chip size, power consumption, performance) not only for aerospace and military applications but also for industry and even consumer electronics. The constant trend of cost and schedule reduction in microelectronics hardware design and development makes it reasonable to develop customized computing systems for particular applications and gives new momentum to the market of embedded systems. Such systems consist of a dedicated hardware platform developed for a particular application and a problem-specific software optimized for that hardware.

The process of simultaneous design and development of hardware and software components of an embedded system is usually referred to as *hardware/software codesign and codevelopment*. This broad term covers a number of subprocess or activities related to embedded system creation:

1. design phase, including functional design, when requirements are studied and transformed into functional architechture, and hardware/software partitioning, when functions are divided between hardware and software components;
2. development phase or software/hardware codevelopment when both hardware and software teams develop their components; both development activities may influence each other;
3. verification; it spans from unit and module tests to early integration testing in simulator/emulator.

Pakulin N. and Rubanov V. (2009).
Early Creation of Cross Toolkits for Embedded Systems.
In *Proceedings of the International Workshop on Networked embedded and control system technologies: European and Russian R&D cooperation*, pages 108-119

Hardware/software codesign and codevelopment are crutial factors for success of embedded systems. They reduce time-to-market by better parallelization of the development wrokflows, and improve the quality by enabling early identification of design flaws and optimization the performance of the product.

Cross toolkits play an important role in hardware/software codesign and codevelopment. Primary components of such cross toolkits are assembler, linker, simulator, debugger, and profiler. Unlike chip production, development of cross toolkits does not require precise hardware design description; it is sufficient to have just a high-level definition of the target hardware platform: the memory/register architecture and the instruction set with timing specification. This allows developing cross tools as soon as the early design stages even if the detailed VHDL/Verilog specification is not ready yet. Cross tools could be used in the following scenarios:

- Hardware prototyping and design space exploration (e.g. [4] and [15]) – early development, execution and profiling of sample programs allows study and estimation of the overall design adequacy as well as efficiency of particular design ideas such as adding/removing instructions, functional blocks, registers or whole co-processors.
- Early software development including development, debugging and optimizing the software *before* the target hardware production.
- Hardware design validation. The developed cross-simulator could be used to run test programs against VHDL/Verilog-based simulators. This capability could not be overestimated for the quality assurance before the actual silicon production.

1.1 Paper Overview

In this paper, we present a new approach to cross toolkit development to be used in hardware/software co-development environments. The method enables software developers to create the cross tools as early as the system design phase, to follow rapidly hardware design changes, most notably instruction set modifications, thus reducing the overall time frame of the design phase.

The article is organized as follows. Section 2 discusses generic requirements to cross toolkit develpment that hardware/software codevelopment imposes. Section 3 presents the new ADL language for defining instruction set called *ISE*. Section 4 introduces MetaDSP framework for cross toolkit development that uses hybrid hardware description with both high-level ADL part and efficient C/C++ part. Section 5 briefly overviews several industrial applications of ISE and MetaDSP framework. Conclusion summarizes the lessons learned and gives some perspectives for future development.

2 Hardware/Software Codesign and Codevelopment Requirements to Cross Toolkit Development

Let's consider a typical co-development process depicted at the fig. 1. The development process involves at least two teams - one is working on the hardware part of the system while another one focuses on software development.

2.1 Related Work

Efficient cross toolkit development process requires automation to minimize time and effort necessary to update the toolkit to match new requirements. Such automation is built around a machine-readable definition of the target hardware platform. There are three groups of languages suitable for this task:

- Hardware Definition Languages (HDL, [10]) used for detailed definition of the hardware;
- Architecture Description Languages (ADL, [9] and [13]) used for high-level description of the hardware;
- and general purpose programming languages (such as C/C++).

HDL specifications define CPU operations with very high level of detail. All three major modern HDL – VHDL [5], Verilog [6], and SystemC [7] – have execution environments that can serve as a simulator to run any assembly language programs for the target CPU: Synopsys VCS, Mentor Graphics ModelSim, Cadence NC-Sim and other. Still, low performance of HDL-based simulators is one of the major obstacles for HDL application in cross toolkit development. Another issue is the late moment of HDL description availability: it appears after completing the instruction set design and functional decomposition. Furthermore, HDL does not contain an explicit instruction set definition that makes automated assembler/disassembler development impossible. These issues prevent from using HDL to automate cross toolkit development.

Architecture Description Languages (such as nML[1], ISDL[2], EXPRESSION[3]) are under active development during the recent decade. There are tools created for rapid hardware prototyping at the high level including cross toolkit generation. Corresponding approaches are really good for early design phase since they help to explore key design decisions. Unfortunately, at the later design stages details in an ADL description become smaller, the size of the description grows and sooner or later it comes across the limitations of the language. As a result, is breaks the efficiency of the simulator generated from the ADL description and makes the profiler to give only rough performance estimates without clear picture of bottlenecks. Cross toolkits completely generated from an ADL description are not applicable for industrial-grade software development yet.

Manual coding with C or C++ language gives full control over all possible details and allows creation of cross toolkits of industrial quality and efficiency. Many companies offer services on cross toolkit development in C/C++ (e.g. TASKING, Raisonance, Signum Systems, ICE Technology, etc.). Still it requires significant efforts and (what is more important) time to develop the toolkit from scratch and maintain it aligned with the requirements. Long development cycle makes it almost impossible to use cross toolkits developed in C/C++ for hardware prototyping and design space exploration.

3 ISE Language

We developed ISE (*I*nstruction *S*et *E*xtension) language to specify hardware design elements that are subject to most frequent changes: memory architecture and CPI instruction set. ISE description is used to generate assembler and disassembler tools completely and to generate components of the linker, debugger and simulator tool.

The following considerations guided the language design:

- the structure of an ISE description should follow the typical structure of an instruction set reference manual (like [14] or [12]) that usually serve as the input for the ISE description development;
- support for irregular encoding of instructions typical for embedded DSP applications including support for large number of various formats, distributed encoding of operands in the word, etc.;
- operational definition of data types, logic and arithmetic instructions, other executable entities should be specified in a C-like programming language.

ISE module consists of 7 sections:

1. **.architecture** defines global CPU architecture properties such as pipeline stages, CPU resources (buses, ALUs, etc.), initial CPU state;
2. **.storage** defines memory structure including memory ranges, I/O ports, access time;
3. **.ttypes** and **.otypes** define data type to represent registers and instruction operands;
4. **.instructions** defines CPU instruction set (see 3.1);
5. **.aspects** defines various aspects of binary encoding of CPU instructions or specifies additional resources or operational semantics of instructions;
6. **.conflicts** specifies constraints on sequential execution of instructions such as potential write after read register or bus conflict; assembler uses conflict constraints to automatically insert NOP instructions to prevent conflicts during software execution.

3.1 Instruction Definition

.instruction section is the primary section an ISE module. It defines the instruction set of the target CPU. For each instruction cross toolkit developers can specify:

- mnemonics and binary encoding;
- reference manual entry;
- instruction properties and resources used;
- instruction constraints and inter-instruction dependencies;
- definition of execution pipeline stage.

Mnemonics part of an instruction definition is a template string that specifies fixed part of mnemonics (e.g. `ADD`, `MOV`), optional suffixes (e.g. `ADDC` or `ADDS`) and operands. A singe instruction might have several definitions depending on the operand types. For example, `MOV` instruction could have different definitions for register-register operation, register-memory and memory-memory operations.

Binary encoding is a template that specifies how to encode/decode instructions depending on the instruction name, suffixes and operands.

Reference manual entry is a human-readable specification of the instruction.

Properties and resources specify external aspects of the instruction execution such as registers that it reads and writes, buses that the instruction accesses, flags set etc. This information is used to detect and resolve conflicts by the assembler tool. Besides this the

instruction definition might specify explicit dependencies on preceding or succeeding instructions in the constraints and dependencies section.

ISE language contains an extension of C programming language called ISE-C. This extension is used to specify execution of the operation on each pipeline stage. ISE-C has extra types for integer and fixed point arithmetic of various bit length, new built-in bit operators (e.g. shift with rotation), built-in primitives for bit handling. ISE-C has some grammar extension for handling operands and optional suffixes in mnemonics. Furthermore ISE-C expression can use a large number of functions implemented in ISE core library.

An example of instruction specification is presented at figure 2.

Please note that unlike classic ADL languages ISE specification does not provide the complete CPU model. The purpose of ISE is to simplify definition of the elements that are subject to the most frequent changes. All the rest of the model is specified using C/C++ code. This separation allows for flexible and maintainable hardware definition along with high performance and cycle-precise simulation.

4 Application to the Codevelopment Process

The proposed hybrid ADL/C++ hardware definition is supported by the *MetaDSP* framework for cross-toolkit development. The framework is intended for use by software developers. Typical use case is as following:

1. hardware developers provide the software team with hardware definition in the form of ISE specification;
2. software developers generate cross tools from the specification;
3. software team develops the software in Embedded-C[8] and build using the generated cross-assembler and cross-compiler;
4. the machine code is executed and profiled in simulator.

To support this use case the framework includes:

- ISE translator that generates components of cross tools from the ISE specification;
- pre-defined components for ISE development (e.g. ISE-C core functions library);
- an IDE for hardware definition development (in ISE and C++), target software development (in Embedded C and assembly languages), controlled execution within simulator; the Embedded C compiler supports a number of optimizations specific for DSP applications[11].

The figure 3 presents the structure of the MetaDSP framework.

MetaDSP toolkit uses ISE specification to generate cross tools and components. For example, the MetaDSP tools generate assembler and disassembler tools completely from the ISE specification. For linker MetaDSP generates information about instruction binary encodings, instruction operands and relocatable instructions. Debugger and profiler use memory structures and operand types from the ISE specification.

The cycle-precise simulator is an important part of the toolkit. Figure 4 presents its architecture. MetaDSP tools generate several components from the ISE specification: memory implementation (from `.storage` section), resources (from `.architectu-`

```
/*
 * This is a C-style block comment.
 */
// This is a C++-style one-line comment.
// <ALU001> - the identifier of the definition.
// ADD[S:A][C:B]  - instruction mnemonics with
// optional parts. Actually defines 4 instruc-
// tions: ADD, ADDS, ADDC, ADDSC.
//  GRs, GRt - identifiers of a general-purpose
//  register. Rules for binary encoding of GRs
//  and GRt are defined in .otypes section.
<ALU001>  ADD[S:A][C:B] {GRs}, {GRt}
 // Binary encoding rule.
 // For example, "ADDC R0, R1" is encoded as
 // 0111-0001-1000-1001
 0111-0A0B-1SSS-1TTT
 // The reference manual string.
 "ADD[S][C] GRs, GRt"
 // instruction properties:
 //  reads the registers GRs and GRt,
 //  writes the register GRs.
 properties [ wgrn:GRs, rgrn:GRs, rgrn:GRt ]
 // Operation of the EXE pipeline stage
 // specifies using ISE-C language.
 action {
  alu_temp = GRs + GRt;
  // If the suffix 'C' is set in mnemonics
  // use 'getFlag' function from the core
  // library.
  if (#B) alu_temp += getFlag(ACO);
  // If the suffix 'S' is set in mnemonics
  // use 'SAT16' function from the core
  // library.
  if (#A) alu_temp = SAT16(alu_temp);
  GRs = alu_temp;
 }
```

Fig. 2. An example of instruction specification.

`re` section), instruction implementations and decoding tables (from `.instruction` section), as well as conflicts detector and instruction metadata.

Within the presented approach certain components are specified in C++:

- control logic, including pipeline control (if any), address generation, instruction decoder;
- memory control;
- model of the peripheral devices including I/O ports.

- 16-bit RISC DSP CPU with support for Adaptive Multi-Rate (AMR) sound compression algorithm. Produced 25 major releases of the cross-toolkit.
- 32-bit RISC DSP CPU with support for Fourier transform and other DSP extensions. Produced 39 major releases of the cross-toolkit.
- 16/32-bit RISC CPU clone of ARM9 architecture.
- 16/32-bit VLIW DSP CPU with support for Fourier transform, DMA, etc. Produced 33 major releases of the cross-toolkit.

The following list summarizes lessons learned from the practical applications of the approach. We compared time and effort needed in a pure C++ development cycle of cross toolkits with the ISE-enabled process:

- size of assembler, disassembler and simulator sources (excluding generated code), in lines of code: reduced by 12 times;
- cross-toolkit development team (excluding C compiler development): reducing from 10 to 3 engineers;
- number of errors detected in the presentation of hardware specifications in cross tools: reduction by the factor of more than 10;
- average duration of the toolkit update: reduced from several days to hours (even minutes in many cases).

5.1 Performance Study

This section presents a performance study of a production implementation of the AMR sound compression algorithm. The study was performed on Intel Core 2 Duo 2.4 GHz.

The size of the implementation was 119 C source files and 142 C header files, and 25 files in the assembly language; total size of sources was 20.2 thousand LOC without comments and empty lines. The duration of the audio sample (10 seconds voice speech) lasted 670 million of cycles on the target hardware.

Table 1 presents elapsed time measurements of the generated cross tools for the AMR case study. Table 2 presents measurements of the generated simulator in MCPS (millions of cycles per second).

Table 1. AMR sample – cross toolkit performance.

Operation	Duration, sec.
Translation (.c → .asm)	22
Assembly (.asm → .obj)	14
Link (.obj → .exe)	1
Build, total	**37**
Execution on the audio sample (fast mode)	53
Execution on the audio sample (debug mode with profiling)	93

Table 2. AMR sample – simulator performance.

Execution mode	MCPS
Fast mode	12.6
Debug mode with profiling	7.2
Peak performance on a synthetic sample	25.0

6 Conclusions

The paper presents an approach to automation of cross toolkit development for special-purpose embedded systems such as DSP and microcontrollers. The approach aims at creation the cross tools, namely assembler/disassembler, linker, simulator, debugger, and profiler, at early stages of system design. Early creation of the cross tools gives opportunity to prototype and estimate efficiency of design variations, co-development of the hardware and software components of the target embedded system, and verification and QA of the hardware specifications before silicon production.

The presented approach relies on a two-level description of the target hardware: description of the most flexible part – the instruction set and memory model – using the new ADL language called *ISE* and description of complex fine grained functional aspects of CPU operations using a general purpose programming language (C/C++). Having ADL descriptions along with a framework to generate components of the target cross toolkits and common libraries brings high level of responsiveness to frequent changes in the initial design that are a common issue for modern industrial projects. Using C/C++ gives cycle-accurate simulation and overall efficiency of the cross toolkits that meets the needs of industrial developers. The approach is supported by a family of tools comprising MetaDSP framework.

The approach is applicable to various embedded systems with RISC core architectures. It supports simple pipelines with fixed number of stages, multiple memory banks, instructions with fixed and variable cycle count. These facilities cover most of modern special purpose CPUs (esp. DSP) and embedded systems. Still some features of modern general purpose high performance processors lay beyond the capabilities of the presented approach: superscalar architectures, microcode, instruction multi-issue, out-of-order execution. Besides this, the basic memory model implemented in MetaDSP does not support caches, speculative access, etc.

Despite the limitations of the approach mentioned above it was successfully applied in a number of industrial projects including 16 and 32-bit RISC DSPs and 16/32 ARM CPUs. Number of major design changes (with corresponding releases of cross toolkits) ranged in those projects from 25 to 40. The industrial applications of the presented approach proved the concept of using the hybrid ADL/C++ description for automated development of cross toolkits in a volatile design process.

References

1. Fauth, A., Praet, J. V., and Freericks, M. (1995). Describing instruction set processors using nML. In *Proc. of European Design and Test Conference*.
2. Hadjiyannis, G., Hanono, S., and Devadas, S. (1997). ISDL: An instruction set description language for retargetability. In *Proc. of 34th Design Automation Conference*.
3. Halambi, A., Grun, P., Ganesh, V., Khare, A., Dutt, N., and Nicolau, A. (1999). EXPRESSION: A language for architecture exploration through compiler/simulator retargetability. In *Proc. of European Conference on Design, Automation and Test*.
4. Hartoog, M., Rowson, J., and Reddy, P. (1997). Generation of software tools from processor descriptions for hardware/software codesign. In *Proc. of Design Automation Conference (DAC)*.
5. IEEE Std 1076-2000 (2000). VHDL language reference manual.
6. IEEE Std 1364-2005 (2005). Hardware description language based on the Verilog hardware description language.
7. IEEE Std 1666-2005 (2005). System C language reference manual.
8. ISO/IEC TR 18037:2008 (2004). Programming languages – C – Extensions to support embedded processors.
9. Mishra, P. and Dutt, N. (2005). Architecture description languages for programmable embedded systems. In *IEEE Proceedings Computers and Digital Techniques*, volume 152-3.
10. Navabi, Z. (2007). *Languages for Design and Implementation of Hardware*. The VLSI Handbooks. CRC Press, 2nd edition.
11. Rubanov, V., Grinevich, A., and Markovtsev, D. (2006). Specific optimization features in a C compiler for DSPs. In *Programming and Computing Software*, volume 32-1, pages 19–30.
12. Texas Instruments (2006). TMS320C6000 CPU and instruction set reference guide.
13. Tomiyama, H., Halambi, A., Grun, P., Dutt, N., and Nicolau, A. (1999). Architecture description languages for systems-on-chip design. In *Proc. of Asia Pacific Conf. on Chip Design Language*, pages 109–116.
14. VIA Technologies (2005). MicroDSP 2 instruction set description.
15. Yung-Chia, L. (2003). Hardware/software co-design with architecture description language.