ISE language: the ADL for Efficient Development of Cross Toolkits

Nikolay Pakulin and Vladimir Rubanov

Institute for System Programming of the Russian Academy of Sciences,
Moscow, Russia,
npak@ispras.ru vrub@ispras.ru

Abstract. Cross toolkits (assembler, linker, debugger, simulator, profiler) are widely used for software-hardware codesign; an early creation of cross toolkits is an important success factor for industrial embedded systems. At the hardware design stage systems are subject to significant design alterations including changes in the instruction set of target CPUs. This is a challenging issue for early cross toolkit development. In this paper, we present a new Architecture Description Language (ADL) called *ISE language* and an approach to early cross toolkit development to cope with hardware design changes. The paper introduces the MetaDSP framework that supports ISE-based construction of cross toolkits and gives brief overview of the MetaDSP applications to industrial projects that proves the industrial strength of the presented approach and tools.

1 Introduction

Nowadays we witness creation of various embedded systems with rather strict 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 special-purpose computing systems for various applications and gives new impulse 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.

Cross tools play an important role for bringing an embedded system to life as they allow development, debugging and profiling of the target software on powerful workstations which do not suffer from the limitations of the target embedded systems and typically run on CPUs which architecture and instruction set are different from the target CPUs. 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 high-level definition of the target hardware platform: the memory/register architecture and the instruction set with cycle specification. This allows developing cross tools as soon as the early design stages even if exact VHDL/Verilog specification is not ready yet. Such co-development has the following crucial benefits:

– Hardware prototyping and design space exploration (e.g. [1] and [2]) – 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. It reduces time-to-market for the complete "Hardware + Software" product.
– 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 actual silicon production.

The first feature mentioned above – design space exploration – results in frequent changes of requirements. System designers may decide to add or remove an instruction or modify the register file of the CPU. Cross toolkit developers must rapidly answer to such changes and produce new version of the toolkit in short terms. Besides, this practice imposes certain quality and performance requirements on the cross toolkits and on the simulator in particular. Special attention should be paid to the performance efficiency of the simulator.

1.1 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, [3]) used for detailed definition of the hardware;
– Architecture Description Languages (ADL, [4] and [5]) 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 [6], Verilog [7], and SystemC [8] – 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[9], ISDL[10], EXPRESSION[11]) 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.

1.2 Paper Overview

In this paper, we present a new approach to cross toolkit development that combines the power of high-level definition using ADL-like language and the efficiency of the modern programming languages. The method provides reasonable level of automation with support for rapid requirement changes and co-development of hardware and software components of modern embedded systems.

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

2 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 [12] or [13]) 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 2.1);
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.

2.1 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 1.

```
/*
 * 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 instructions: 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. 1. An example of instruction specification.

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.

3 Development Process

The proposed hybrid ADL/C++ hardware definition is supported by the *MetaDSP* framework for cross-toolkit development. 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[14] and assembly languages), controlled execution within simulator; the Embedded C compiler supports a number of optimizations specific for DSP applications[15].

Figure 2 presents the structure of the MetaDSP framework.

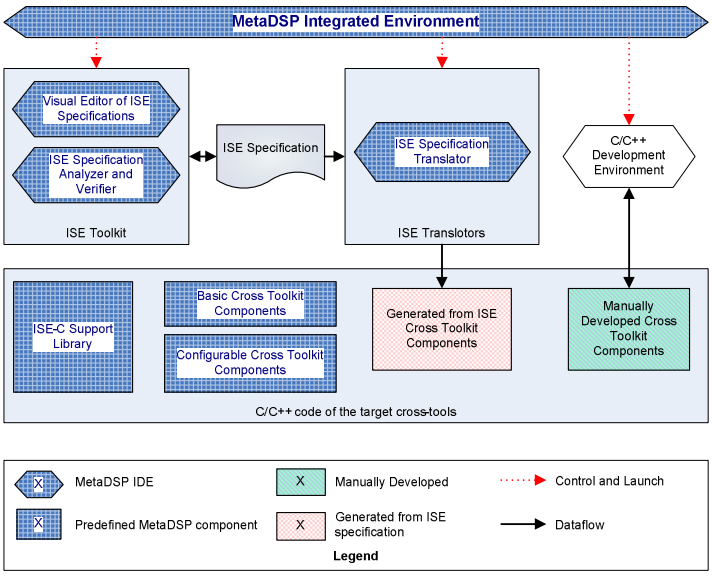

Fig. 2. MetaDSP framework structure

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 3 presents its architecture. MetaDSP tools generate several components from the ISE specification: memory implementation (from `.storage` section), resources (from `.architecture` section), instruction implementations and decoding tables (from `.instruction` section), as well as conflicts detector and instruction metadata.

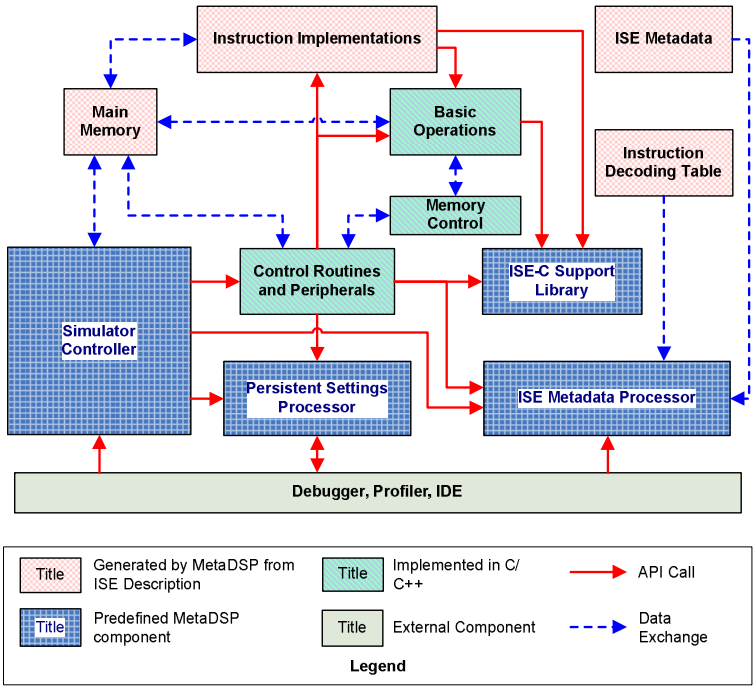

Fig. 3. MetaDSP simulator architecture

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.

For most of the manual components MetaDSP tools generate stubs or some basic implementation in C++. Developers may use the generated code to implement peculiarities of the target CPU, such as jumps prediction, instruction reordering, etc.

Using C/C++ to implement CPU control logic and memory model facilitates high performance of the simulator. Another benefit of using C/C++ compared to true ADL languages is an early development of the cross toolkit: it might start before completing the function decomposition of the target CPU; thus the simulator could be used to experiment with design variations.

Figure 4 presents the snapshot of OSCAR Studio, the IDE for MetaDSP framework.

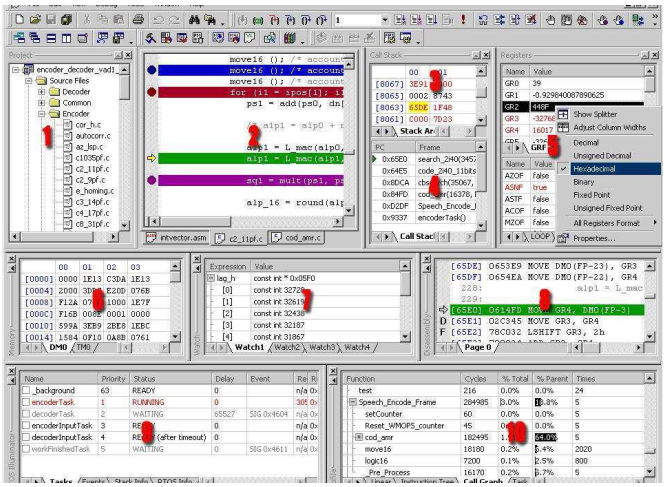

Fig. 4. OSCAR Studio: the IDE for MetaDSP framework

Red numbers mark various windows of the IDE:

1. Project Navigator window. It displays the tree of the source files and data files.
2. Source Code Editor window. The editor supports syntax highlight and instruction autocompletion (from the ISE specification). The editor window is integrated with the debugger - it marks break points, frame count points and trace points.
3. Stack Memory window displays the contents of the stack.
4. Call Stack window displays the enclosing frames (both assembly subroutines and C functions).
5. Register window displays the contents of the CPU registers.
6. Memory dump window displays contents of various memory regions.

7. Watch window displays the current value of arbitrary C expressions.
8. Code Memory window displays the instructions being executed. It supports both binary and disassembly forms as well as displaying the current pipeline stage (fetch, decode, execute, etc.).
9. OS debugger window displays the current state of the execution environment (OS): list of the current tasks, semaphores, mutexes, etc.
10. Profiler window displays various profiling data. The profiler is integrated with the editor window as well – the editor can show profiling information associated with code elements.

4 Industrial Applications

The approach presented in this paper and MetaDSP framework were applied to five industrial projects. Please note that the each "major releases of the cross toolkit" mentioned in the project list below is caused by a major change in CPU design such as modification of the instruction set or memory model alteration.

- 16-bit RISC DSP CPU with fixed point arithmetic. Produced 25 major releases of the cross-toolkit.
- 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).

4.1 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

5 Conclusion

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. M. Hartoog, J. Rowson, P. Reddy. Generation of Software Tools from Processor Descriptions for Hardware/Software Codesign. Design Automation Conference (DAC) 1997.
2. Lin Yung-Chia. Hardware/Software Co-design with Architecture Description Language. Programming Language Lab. NTHU. 2003.
3. Z. Navabi. Languages for Design and Implementation of Hardware. The VLSI Handbook2nd ed. CRC Press, 2007.
4. P. Mishra and N. Dutt. Architecture description languages for programmable embedded systems. IEEE Proceedings Computers and Digital Techniques. Vol. 152, No. 3, May 2005.
5. H. Tomiyama, A. Halambi, P. Grun, N. Dutt, A. Nicolau. Architecture Description Languages for Systems-on-Chip Design. Proc. Asia Pacific Conf. on Chip Design Language, 1999, pp. 109116.
6. VHDL Language Reference Manual. IEEE Std 1076-1987.
7. Hardware Description Language Based on the Verilog Hardware Description Language. IEEE Std 1364-2005.
8. System C Language Reference Manual. IEEE Std 1666-2005.
9. A. Fauth, J. Van Praet, M. Freericks. Describing instruction set processors using nML. In Proc. of EDTC, 1995.

10. G. Hadjiyannis, S. Hanono, S. Devadas. ISDL: An Instruction Set Description Language for Retargetability. Design Automation Conference (DAC) 1997.
11. A. Halambi, P. Grun, V. Ganesh, A. Khare, N. Dutt and A. Nicolau. EXPRESSION: A Language for Architecture Exploration through Compiler/Simulator Retargetability. DATE 99.
12. MicroDSP 2 Instruction Set Description. VIA Technologies Manual, 2005.
13. TMS320C6000 CPU and Instruction Set Reference Guide. Texas Instruments Literature Number SPRU189F. http://focus.ti.com/lit/ug/spru189g/spru189g.pdf.
14. ISO/IEC TR 18037:2008. Programming languages – C – Extensions to support embedded processors. 2004.
15. V. Rubanov, A. Grinevich, D. Markovtsev. Programming and Computing Software Vol. 32-1, pp. 19-30, 2006