A Design and Tool Reuse Methodology for Rapid Prototyping of Application Specific Instruction Set Processors

Young Geol Kim
Dept. EE, Korea Advanced Institute of Science and Technology
Gusong-dong, Yusong-gu , Taejon, KOREA
ygkim@smslab.kaist.ac.kr

Tag Gon Kim
Dept. EE, Korea Advanced Institute of Science and Technology
Gusong-dong, Yusong-gu , Taejon, KOREA
tkim@smslab.kaist.ac.kr

Abstract

This paper proposes a design and tool reuse schemes for a rapid prototyping of application specific instruction-set processors (ASIP). We propose a three-level hierarchical architecture abstraction method for top-down processor design. We also propose a reusable architecture description language (READ) and a family of retargetable simulators that allow a top-down processor description and prototyping from instruction-set design to RTL implementation.

1 Introduction

As the time-to-market requirement gets stronger in processor development, a rapid prototyping methodology becomes more important than ever. Our approach to this problem assumes that the total design time in processor development might be significantly reduced when we fully reuse tools and design itself. Towards this, we developed a reusable architecture description language, called READ, and a family of retargetable simulators. Section 2 proposes a hierarchical architecture abstraction method along with its description language, READ. Section 3 presents a family of retargetable simulators, while section 4 gives an example of 3D graphics processor description and prototyping with READ and retargetable simulators. Finally, section 5 concludes with further works.

2 Hierarchical architecture abstraction

We propose a hierarchical abstraction of architecture in three levels: *HiISA, LowISA* and *MicroA*. HiISA (High-level Instruction Set Architecture) denotes micro-architecture independent level of (conventional) ISA including instruction syntax/semantics, addressing modes and simple storage description. On the contrary, LowISA (Low-level ISA) describes a micro-architecture dependent part of ISA such as instruction timing and concurrency information for micro-parallel architectures. Finally, MicroA is an abstraction level at which target instruction set is implemented with control path and data path such as cache, bus and pipelined functional units, and so on.

The LowISA can be thought of as an abstraction of lower-level micro-architecture details. For example, we can associate each instruction with a result latency that specifies the number of clock cycles expected for a functional unit to produce a result (at LowISA), without specifying the detailed implementation of the functional unit, which can be developed in later design cycle (at MicroA).

2.1 READ : Reusable Architecture Description

There have been a lot of researches towards describing processor architectures: nML [1], ISPS [3] and MIMOLA [4]. Among them, the nML formalism is one of the most recent and well-known approaches. It describes an instruction-set based processor in a well-defined manner. The nML was applied to generate instruction-set simulators and assemblers/disassemblers for ARM7 and TMS320C50 processor as described in [2]. While the nML can be suited well for describing instruction-set level architectures, it lacks lower-level architecture description capabilities such as delayed assignments, interrupts, improved arithmetic support, and so on. To the contrary, ISPS and MIMOLA do not support high-level construct for instruction-set description as nML does. Instead, they have an expression power of describing detailed architectural information such as processor netlist or decoder logic.

To summarize, there arises a difficulty in using any above-mentioned description languages alone for top-down processor design, since they do not support a seamless description method from instruction-set to RTL design of a processor architecture. As an alternative, we propose a reusable

architecture description language called READ (REusable Architecture Description) that supports a seamless architecture description from instruction-set to micro-architecture. Moreover, since the nML is not designed to directly support architecture changes, an architecture modification can only be done with manual rewriting of nML description, which is error prone and hard to maintain. To solve this problem the READ supports two-dimensional description reuse. It is designed so that proposed hierarchies of architecture can be described well. Figure 1 shows the concepts READ gives throughout the design trajectory.

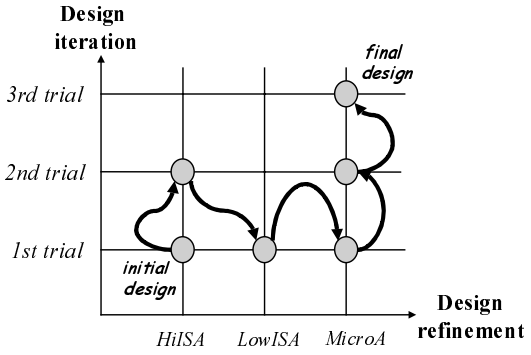

Figure 1: Reusable top-down architecture description language (READ)

Design iteration at level L designates the process of redesigning architecture at level *L*. For example, design iteration at the level of HiISA can include activities such as adding/removing instruction to/from the original instruction set, modifying storage parameters such as the number of registers in the register file. *Design refinement from level L to L+1* is defined as the process of adding architectural details *at lower level L+1* without changing the design *at the higher level L*. For example, adding instruction timing to each instruction described at the level of HiISA could be one of activities in design refinement from HiISA to LowISA. The READ consists of three sub-languages: HiREAD, LowREAD and MicroREAD that describe architectural information of HiISA, LowISA and MicroA, respectively.

2.1.1 Distinguishing features of the READ

Firstly, the READ has syntax for representing design alternatives unlike other description languages mentioned above. The design alternatives constitute a design space from which a design point (specific architecture) is selected for evaluation. Therefore, automatic design space exploration becomes possible with the READ. Secondly, the READ is modular in a sense that every architectural entity is described and managed in a separate file. This modularity enlarges reusability since a modification of some entities does not influence other parts. With the same reasoning, we can easily see that the modularity helps reducing unwanted error propagation caused by non-modular description. Thirdly, the READ borrowed some important concepts from object-oriented paradigm for reusability: inheritance mechanism (white-box reuse) and encapsulation (a separation of interface and implementation). Lastly, the READ supports a step-wise architecture refinement from HiISA to MicroA by partitioning it into three independent sub-languages: HiREAD, LowREAD and MicroREAD. The reusability arising from this sub-division of architecture description could be maximized when the READ is applied to a family of processor implementations with the same instruction-set. For example, if the READ were used to describe an intel x86 family, the HiREAD description could be completely reused.

2.2 HiREAD and LowREAD: ISA description

The HiREAD and LowREAD provide syntax and semantics for HiISA and LowISA description of target processor, respectively. As previously mentioned, the HiREAD mainly describes an instruction-set, addressing modes and storages. An important part of the grammar defined by HiREAD is shown as a BNF form as follows:

HiREAD_description:
Instruction_set_description
| Instruction_description
| Addressing_mode_description
| Storage_description

Instruction_set_description:
Instruction_subset_description
| Instruction_elements_description

Instruction_subset_description:
Instruction_set_description

Instruction_elements_description:
List of instructions in a instruction set

Instruction_description:
Instruction_semantics_description
| Instruction_syntax_description

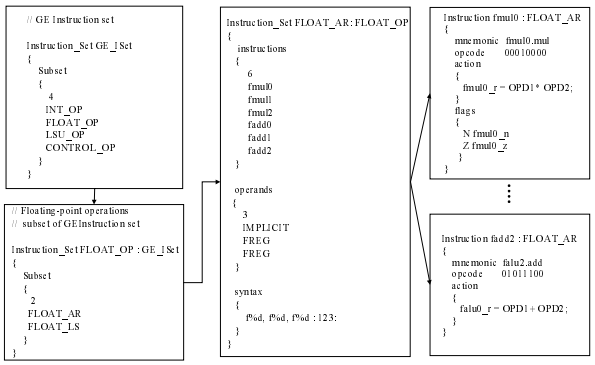

Figure 2: Reuse example in HiREAD

The LowREAD developed so far is only targeted for VLIW architectures. Therefore, its grammar is rather simple since instruction timing information (in cycles) and number of concurrent instructions need only be described. The HiREAD extensively use the inheritance mechanism for description reuse, as shown in Fig.2. In this figure, we can see that the instruction *fmul0* and *fadd2* is a member of instruction set named FLOAT_AR (floating point arithmetic), and they have same operands and syntax description which are described in FLOAT_AR description. In other words, the operands and syntax specification are reused.

2.3 MicroREAD: micro-architecture description

The MicroREAD provides syntax and semantics for architecture description of target processor at micro-architecture level. The notion of *module* is used as a primitive comprising micro-architectural model. It denotes any kinds of hardware object with clear interfaces – input and output ports - and functionality. And, a module can be implemented as behavioral or structural. A structural module specifies its component modules and coupling schemes between them. The MicroREAD does not have syntax for behavior description of a module as yet. Instead, users can describe it in conventional HDL, such as Verilog or VHDL.

These modules are automatically translated into C++ model by MicroARS for simulation (refer section 3.2). Because some kinds of modules are so commonly used in almost every processor that they can be standardized for model reuse. Examples are n-port register file, n-bit register, n-input mux, arbitrary interconnection network, and so on. Our approach is to provide a pre-built library of such standard modules in parameterized C++ classes so that they can be readily used whenever needed. Correspondingly, the MicroREAD also gives a simple and natural method of using standardized modules. An important part of the grammar defined by MicroREAD is shown as a BNF form as follows:

Module_description:
Alternative_description
| Interface_description
| Implementation_description
Interface_description:
Input_port_description
| Output_port_description
Implementation_description:
Behavioral_module_description
| Structal_module_description
Behavioral_module_description:
External_HDL_module_description
| Syntheized_module_description
Structural_module_description:
Component_module_description
| Coupling_description
Component_module_description:
Module_description

Note that a description of components of a module have the same grammar as the enclosing module itself, thereby enabling arbitrary level of hierarchies for module decomposition. The MicroREAD separates the module interface and implementation, thereby more than one module implementations of the same interface can be specified as alternatives. In this way, the module interface can be reused for different module implementations.

3. Tool Reuse : Retargetable Simulators

A family of retargetable simulators, HiRISS (High-level Retargetable Instruction Set Simulator), LowRISS (Low-level Retargetable ISS) and MicroARS (Micro-Architecture level Retargetable Simulator) have been developed to simulate processor architectures at the level of HiISA, LowISA and MicroA, respectively. All versions of retargetable simulators are composed of generic simulation engine on top of which architecture-specific models are attached for simulation. We also developed a family of model synthesizer, which takes the READ description as an input to automatically produce C++ models. The generic simulation engine together with the model synthesizer comprises a retargetable simulator. Fig.3 shows these retargetable simulators. This family of retargetable simulators provides tool reuse since an architecture modification does not require redevelopment of simulators. We employed various design patterns ([5]) in the development of retargetable simulators to make them reusable and extensible as much as possible.

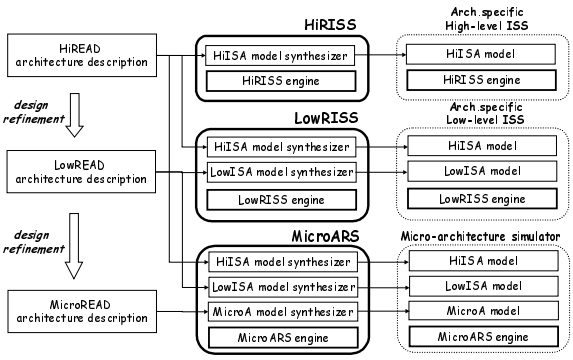

Figure 3: Architecture description and simulator synthesis

3.1 HiRISS/LowRISS : A family of retargetable instruction set simulators

The HiRISS is used to check the functional correctness of developed compiler and assembler for the designed instruction set. Moreover, it can be used to estimate the quality of instruction set by collecting instruction mix (dynamic in-

struction count) with given benchmarks. The LowRISS, retargetable simulator for LowISA simulation, provides rough but rapid estimation of performance parameters such as cycle counts to execute a set of application benchmarks. There is speed/accuracy tradeoff between LowRISS and MicroARS because the LowRISS only requires abstract information about micro-architecture such as instruction timing. The users can not observe detailed behavior that affects performance such as cache miss and resource conflicts with the LowRISS. However, it can be used to obtain quick measurement of overall performance to give guidelines for a micro-architecture design.

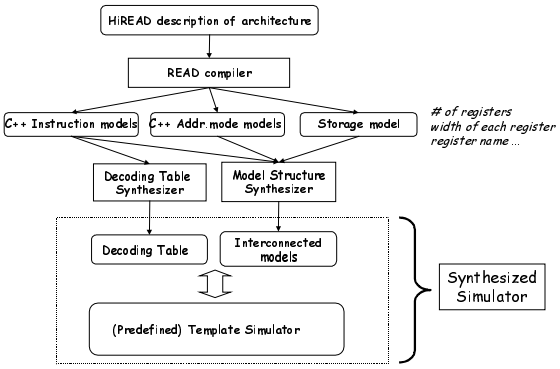

Figure 4: Retargeting mechanism of HiRISS

Fig.4 shows retargeting mechanism of HiRISS from HiREAD in more detail. The generic simulation engine of HiRISS is composed of control path such as Fetcher, Decoder and Executor along with parameterized storage elements such as program and data memory, register file and flag. The HiISA-model synthesizer generates a set of C++ models for instruction, addressing modes and storage. These models are interconnected as described in HiREAD by Model-structure synthesizer. Furthermore, the Decoding table synthesizer generates a decoding table with which the simulator finds appropriate instruction for a given mnemonic or opcode during simulation. The LowRISS further requires a retargetable scheduler for its simulation engine since each instruction now has its own timing information. Although it is required that the LowRISS can handle any kinds of architecture including VLIW and Superscalar, we determined that a different version of LowRISS would make the design simple. And, we have developed a LowRISS for VLIW architecture for the time being.

3.2 MicroARS

The MicroARS (Micro-Architecture level Retargetable Simulator) is used to simulate a designed architecture at register-transfer level to check the functionality of component modules and their interactions, measure various utilization of resources such as functional units, buses and registers. Since all interactions between modules can be observed, important performance parameters such as frequency of pipeline stall can be measured with MicroARS.

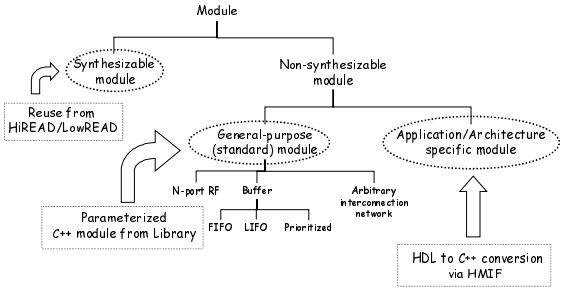

Figure 5: Taxonomy of modules for MicroARS

The MicroARS was designed such that description and model reuse could be fully exploited for retargetability. Figure 5 shows the taxonomy of modules. It divides modules into synthesizable modules and non-synthesizable modules. Non-synthesizable modules are further divided into general-purpose modules and application/architecture specific modules. The examples of synthesizable modules are functional units and decoders. The functional units can be automatically synthesized by reusing each instruction's behavior from HiREAD and timing from LowREAD information. Suppose a functional unit FU can execute several instructions of the given instruction set, then the behavior of the FU is the union of the behaviors of those instructions.

Therefore, in case the designer does not want detailed observation within the FU module, the MicroARS automatically synthesizes the FU module. Otherwise, the designer can model detailed implementation of FU with HDL, which is translated into C++ one later for simulation. Application/architecture specific modules such as floating point adder, floating point multiplier and address generation unit can be modeled in Verilog or VHDL, translated to C++ via hardware module interchange format. To summarize, abstract functional units and general-purpose modules can be reused with MicroARS. The model synthesis capability is the strong point of MicroARS over any kinds of simulators for HDL ever come out.

4 Example : 3D Graphics Processor

This section exemplifies the READ language and the retargetable simulators with 3D graphics processor which executes various geometry processing including geometry transformation, projection, lighting, clipping and triangle setup for rasterization. Fig.6 shows the main data path of

the architecture of graphics processor called GE, which is VLIW type. It consists of

three floating-point multipliers and adders, two load/store units that have a 128-bit bandwidth. It also includes floating point reciprocal unit. The GE has four pipeline stages of *fetch*, *global decoding and contro*l stage, *local decoding* stage, and *execution/write* back stage.

Fig. 6: GE processor architecture: main data path

Since fetch and global decoding stage is architecture-specific and complex, we modeled these components with Verilog HDL at register transfer level. Each functional unit has a local decoder, execution data path, local register file and several implementation of them and did not want to wait until they were developed. Finally, we used ready-made standard modules for demux, latch, register, register file, bus, and data memory. The algorithm we prepared for simulation was geometry transformation algorithm, which is essentially matrix product.

Fig. 7: Transformation algorithm and simulation result

Fig.7 shows the binary code, which is stored in program memory. This code was fed into the MicroARS for simulation at micro-architecture level. The part of simulation result is also shown in Fig.7 (right side). Lastly, Fig.8 shows the *module interface* description of FALU0_Unit (left part), which is one of functional units in GE, and *the module implementation* description (structural) in MicroREAD (right part). Fig.9 shows corresponding C++ model that was automatically generated by MicroARS for simulation.

Fig.8 : Module interface and structural implementation described in MicroREAD

5 Conclusion

This paper proposed a description and tool reuse for rapid prototyping of application-specific processors. The reuse mechanisms we propose are three folds: reuse of architecture description with READ, reuse of tool with retargetable simulators, and model reuse with synthesized module and standard modules. As a further work, we will develop a reusability measure for quantification of reusability. And the proposed framework needs to be tested with more large and complex example. Toward more rapid prototyping, an efficient design space exploration method should be developed.