A Visual Language for Describing Instruction Sets and Generating Decoders

Trevor Meyerowitz, *Student Member, IEEE;* Jonathan Sprinkle, *Member, IEEE;*
Alberto Sangiovanni-Vincentelli, *Fellow, IEEE*

Abstract— We detail the syntax and semantics of ISA_ML, a visual modeling language for describing Instruction Set Achitectures of microprocessors, and an accompanying tool that takes a description in the language and generates decoders from it in the form of a disassembler and a micro-architectural trace interfacer. The language and tool were built using the Generic Modeling Environment (GME), and leverage the concepts of meta-modeling to increase productivity and to provide extensive error checking to the modeler. Using this tool, we were able to construct a model of significant subsets of the MIPS, ARM, and PowerPC instruction sets each in 8 hours or less. This language can be retargeted for other purposes, such as generating synthesizable instruction decoders.

Index Terms— Computer Architecture, Design Automation, Visual Languages, Microprocessors

I. INTRODUCTION

Software is a growing concern in embedded systems design because of the flexibility that software implementations of complex functions offer and because of the stringent constraints placed on timing, memory occupation and quality that they must satisfy. To verify software intensive designs, it is important to accurately model the execution time of the processor(s) in the system. A microprocessor consists of the Instruction Set Architecture (ISA) used to program it, and the underlying microarchitectural implementation of the ISA. For a given ISA (e.g. ARM, MIPS, x86) there are typically a wide variety of microarchitectural implementations with varying degrees of performance (e.g. latency, clock speed, power, area). Designers must either obtain an accurate model from the processor vendor, or construct the model themselves. A less desirable alternative is using a functional Instruction Set Simulator (ISS), which only provides feedback on the number of instructions executed, but execution time can only be inferred in an approximate way. In either case, design space exploration at the micro-architectural level is difficult.

We have developed a methodology and micro-architectural models [11] that interface with functional models to allow for exploration within the Metropolis system-level design framework [4] using formal models of computation to simplify the modeling process. Still, targeting a different instruction set requires several steps. First, the ISS must be modified to output the trace of instructions that it executes. Next, a decoder must be built in order to translate traces into objects that the micro-architectural model can use. Finally, the micro-architectural model is configured to represent the chosen processor.

Our work helps automate the second step of this process by generating the decoding code from a representation described in the visual language ISA_ML (An ISA Modeling Language). This is the most difficult part of the interfacing process because each instruction type must be interfaced to the micro-architectural model, and it is often hard

to reuse code from the ISS to do this. ISA_ML was developed in the GME meta-modeling framework [1]. ISA_ML is a domain specific modeling language for rapidly constructing descriptions of the encodings of Instruction Set Architectures (ISA's) in an intuitive manner consistent with how such ISA's are typically specified in their manuals. To accompany the language, we developed a tool that synthesizes disassembly code and a trace-interface for our micro-architectural models. Furthermore, the tool provides extensive static and dynamic error checking for models.

A. Meta-modeling and Model Integrated Computing

Meta-modeling is a term that implies modeling at a higher level of abstraction, but its precise meaning depends on the context in which it is used. In Metropolis, the meta-model is based on a precise and flexible semantics that can be used to model a wide variety of models of computation. The type of meta-modeling that this paper refers to is a tool for building domain specific modeling environments.

The tool adopted here is GME (the Generic Modeling Environment)[1]. It uses the formalisms of UML (the unified modeling language) and OCL (the Object Constraint Language) to construct domain specific modeling environments [9]. GME follows the Model-Integrated Computing [2] paradigm that allows the user to construct domain specific modeling environments quickly, and serves as a platform for such environments to run on.

B. Related Work

Traditional simulators such as [10][5][3] provide good performance, but are hard to retarget or modify. Simplescalar does provide significant capabilities for micro-architectural design space exploration, but its low-level implementation makes it difficult to modify or extend. Our work is closest to that of Architecture Description Languages (ADL's) such as [6][7][8]. These are specialized languages for describing the microprocessors and their micro-architectures. Among these, Sim-nml [6] is the most similar to our approach in that it takes a structural view to the specification of instruction sets, but it is text-based and less intuitive to learn and use than our language. Expression [8] is also a visual language, but it is primarily targeted at describing pipeline-based micro-architectures, something that we orthogonalize from the ISA. LISA[7] is the most well-known ADL, but it mixes the function and micro-architecture, making reuse difficult.

C. Outline of Paper

The paper is organized as follows: In Section II, we explain the syntax of the ISA_ML language by describing the individual elements of the language and the language rules. Section III illustrates the language semantics with a simple, but detailed example. Section IV presents the design of the interpreter. In Section V we present our results in describing large portions of the MIPS, ARM, and PowerPC ISA's. Section VI wraps up the paper and includes a discussion of future work.

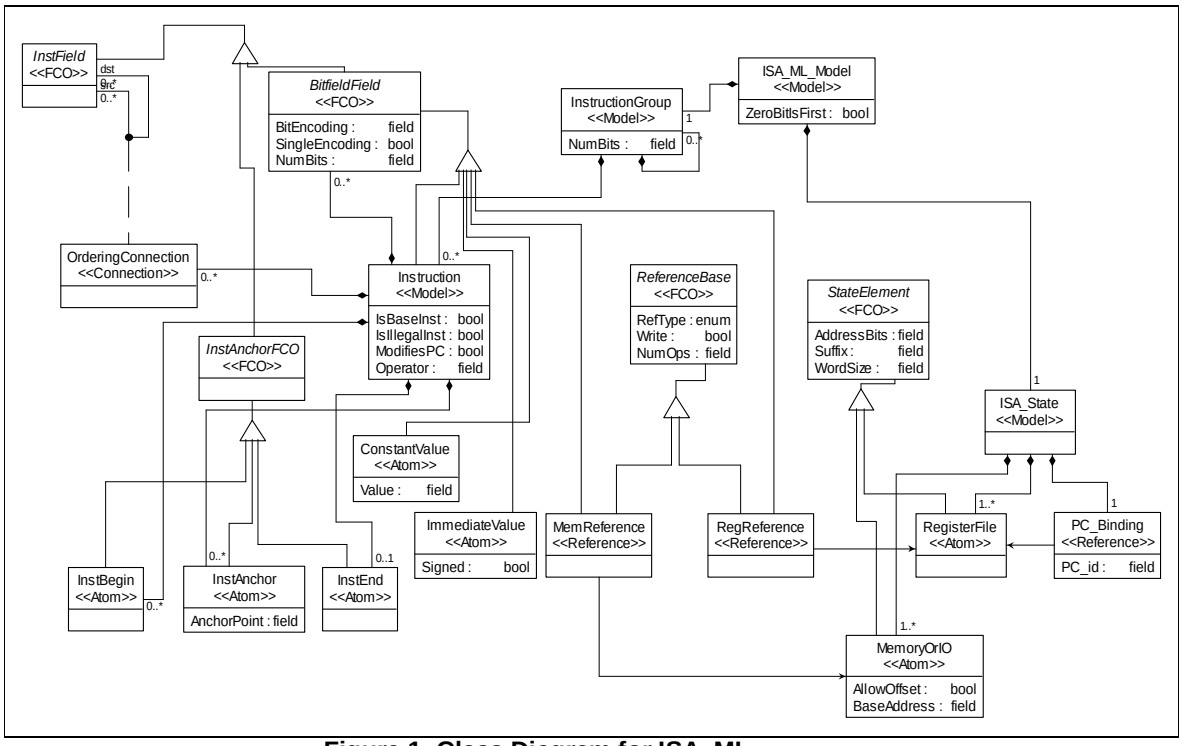

Figure 1: Class Diagram for ISA_ML

II. LANGUAGE SYNTAX

Figure 1 shows the class diagram of the ISA_ML paradigm. A legal ISA_ML model has one ISA_State and a single top level Instruction Group. The instruction set state, *ISA_State* is present on the lower right of the paradigm, and the group of instructions. *InstructionGroup*, is present on the left-hand side with instruction being in the left-center area.

A. Instruction Set State

The ISA state elements are pictured in Figure 2. The memories are for instruction storage, data storage, and in addition, they can represent I/O devices mapped to particular memory addresses. Register files are for data and program counter storage.

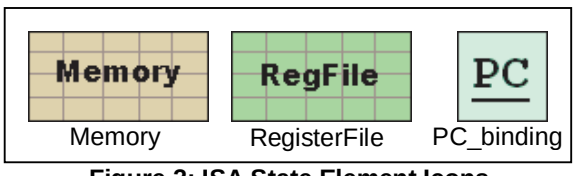

Figure 2: ISA State Element Icons

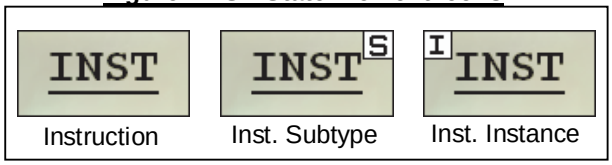

Figure 3: Instruction Icons

Both memory and register files are defined by: the number of bits used to address them, the size of words that they store and a unique suffix. The memory instance also can

allow offsets, and is given a base address. All legal models have at least one memory and at least one register file. There is exactly one *PC_binding* which represents the program counter and must point to a particular register in a register file.

B. Instruction Groups and Instructions

Exactly one top-level instruction group contains all of the definitions of instructions supported by the instruction set. In this version all instructions must be of the same bit-width. Instructions and their operands[1] are both derived from the *Bitfield* type. Bitfields have the following arguments: a non-negative integer *BitLength*, an encoding string called *BitfieldEncoding*, and a Boolean variable *SingleEncoding* that indicates if it has a single encoding or is all don't cares.

1) Instructions

Instructions can be of three types, base instructions, actual instructions, and illegal instructions. Base instructions are used to define families of instructions. Actual instructions are those from the instruction set. Illegal instructions represent encodings that are prohibited in the instruction set. The Boolean variable *ModifiesPC* indicates if the instruction writes to the program counter in a non-standard way (e.g. a branch instruction). The string *Operator* indicates the instruction's operator name. An instruction is a base instruction if the Boolean variable *IsBaseInst* is true. The Boolean *IsIllegal* indicates if the instruction is an illegal instruction. Only instances and subtypes of base instructions can be contained within other instructions. The different icons for the instruction, sub-type, and instance are shown in Figure 3. Actual instructions must be unique in their names and in their encodings that they present. The encodings are specified in two ways: directly at the top level, or via the operands that the instruction model contains.

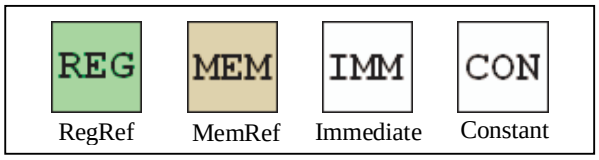

Figure 4: Base Operand Icons

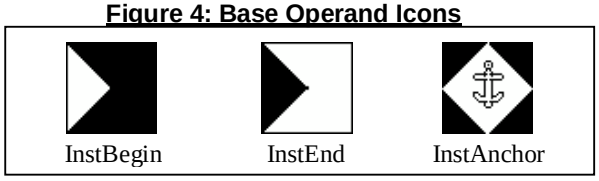

Figure 5: Anchor Icons

2) Base Operands

The types of bit-fields supported within instructions are *RegReference*, *MemReference*, *ImmediateValue*, *ConstantValue*, and other instructions. The first two are references respectively to a register file and a memory/io area. The references are to one or more operands and are read or write. Currently ISA_ML supports single-element references, and Boolean-list references that have one bit for each element in the referenced memory/register element. *ImmediateValue*'s are numbers specified directly in the bitfield and can be signed or unsigned. *ConstantValue* is similar to ImmediateValue, but it cannot have *don't cares* and is specified with an integer that is converted to the appropriate bit-field. Figure 4 shows the base operand icons.

---

[1] Even though an instruction can be an operand of another instruction, we refer to operands as non-instruction operands.

3) Anchors

Instructions can also contain anchors, which serve as marking points of particular bits in the instruction. *InstBegin* and *InstEnd* are anchors that represent the beginning and end bits of the instruction respectively. In addition, a general anchor, *InstAnchor*, can be used where the bit referenced is specified. Anchors appear as ports that can be connected to other instructions and operands. The anchor icons are shown in Figure 5.

4) Connections

Elements can be connected by *ordering connections.* A connection from one operand to another indicates that the first operands bits precede those of the second operand (which the arrow points to). Connections can also be connected to anchors and instructions. In this sub-section, we detail the semantics of all legal connections. A non-instruction object can have at most one incoming and one outgoing connection. In the case of an instruction with anchors, it can have more connections, as long as the instruction itself and the anchors each only having at most one incoming and outgoing connection. Below we list the semantics of the different types of ordering connections.

- **Operand-Operand Connections** - the first bit of the destination operand will be the bit immediately after the last bit of the source operand.
- **Operand-Anchor Connections** – Let *b* be the bit number of the anchor. A connection from the operand to the anchor specifies that the last operand bit is the $(b-1)^{th}$ bit. A connection from the anchor to the operand specifies that the first operand bit is the b^{th} bit.
- **Instruction-Anchor and Operand Connections** – A connection from an external operand to an anchor within an instruction behaves the same as an operand-anchor connection, but within an instruction. If there is a position conflicts between the low-level instruction and its container instruction, the container's position takes precedence.
- **Instruction-Operand Collapsing Connections** – We call a connection between an operand and an instruction a collapsing connection because the operand is "collapsed" into the instruction. If the connection is from the operand to the instruction, then the operand is inserted in the first unoccupied bits of the instruction. If the connection is from the instruction to the operand, then the operand is inserted backwards, where its last bit will occupy the last available bit in the instruction.

C. Language Rules

Below we list the language rules. We indicate when rules are statically checked by constraints, and all others are dynamically checked by running the interpreter. These language rules include:

1. Fully deterministic ordering within an instruction.
2. Each operand can have at most one incoming connection and one outgoing connection.
3. All actual and illegal instructions must have the same bit-width
4. The operands of an instruction must have unique names. (Solved by a constraint)
5. The operands of an instruction must be consistent with the encoding of the instruction[2]
6. Each instruction has a unique name (Solved by a constraint)
7. Each reference operand must be non-null (Solved by a constraint)

[2] (i.e. their bit-field sizes must be less than or equal to the number of "don't cares" (x's) in the instruction's bit field)

8. Only instances and subtypes of base instructions are allowed in other instructions

III. SIMPLE EXAMPLE

Figure 6 shows a summary of the encodings of all of the instructions in this example. *ArithmeticBase* is a base-instruction that forms the basis for the *Addition*, *Subtraction*, and *Multiply-Accumulate* (MAC) instructions. In this example, all of the instructions are 32-bit long, with 4-bit register references, and bit 31 is the first bit in every instruction[3].

	31..30	29...26	25..22	21..18	17..14	13..10	9..8	7..4	3..0
Base	11	xxxx	R_M	R_N	xxxx	xxxx	xx	xxxx	R_D
Add	11	0111	R_M	R_N	xxxx	xxxx	xx	conf	R_D
Sub.	11	0001	R_M	R_N	xxxx	xxxx	00	conf	R_D
MAC	11	0011	R_M	R_N	R_{MAC}	xxxx	xx	xxxx	R_D

Figure 6: Instruction Encoding Summary

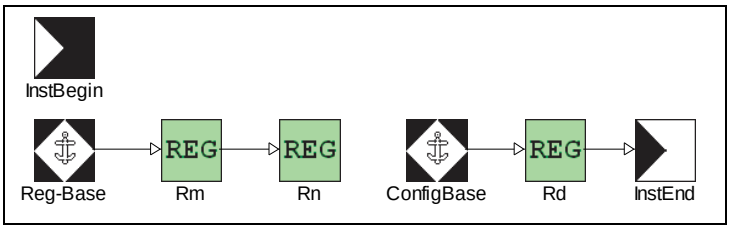

Figure 7: Base Arithmetic Instruction

A. Base Arithmetic Instruction

Figure 7 shows the base instruction that the other instructions are built from. Its encoding is specified as two "1"'s and then 30 "x"'s. It has *InstBegin*, *InstEnd* anchors and two custom anchors *RegBase* and *ConfigBase* that are set at bits 25 and 3 respectively. R_D is a write operand, and R_M and R_N are read operands.

B. Actual Instructions

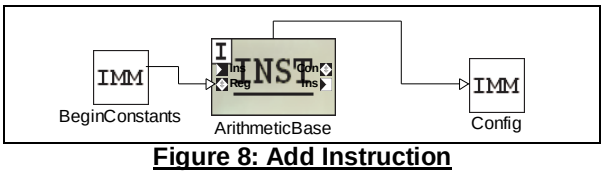

Figure 8: Add Instruction

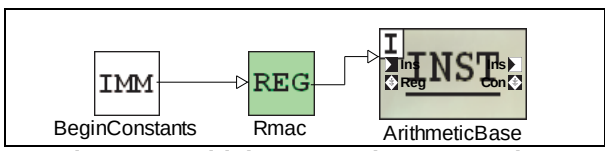

Figure 9: Multiply Accumulate Instruction

Figure 8 shows the definition of the Add Instruction[4]. It has the 4-bit immediate *BeginConstants* with the value of "0111", which is connected directly to the *RegBase* port (bits 29-26). Then there is the 4-bit ImmediateValue *Config* that is has an ordering connection from the instruction to it. This backwards collapsing connection means that

[3] You can also specify in the top-level model that 0 is the first bit.
[4] The subtract instruction has a similar form, and so it is omitted.

the operand occupies the last 4 unspecified bits of the base instruction (i.e. bits 7-4). Figure 9 contains the multiply accumulate instruction. It has two chained operands that are connected to *ArithmeticBase*. This means that they will fill in the un-filled bits starting from the beginning of the instruction. This puts *BeginConstants* at bits 29-26 and R_{MAC} in bits 17-14.

IV. INTERPRETER

The end goal of the interpreter for this project is to create a disassembler/decoder in C++ files that can decode an instruction stream. The interpreter was developed in Visual C++ beginning with the files generated by the GME BON (Builder Object Network) Extender, and utilizes the visitor design pattern for traversing the model. The interpreter first assembles the instruction set and checks that the language rules are followed. After this, it does a consistency check on the instruction encodings. Then it creates a header file to support the decoding of the instructions. Finally, C++ files for disassembling and interpreting a trace of instructions are generated.

A. Instruction Set Assembly

Once the design in ISA_ML passes all constraints the interpreter is run to check other language rules, and then assemble the instruction set. First, the instruction-set state is visited and the register files, memory, and program counter are all verified. After this, the instructions are visited. First, the instructions are split up into lists of actual, illegal, and base instructions. Then, the base instructions are assembled and verified. After this, the actual and illegal instructions are assembled and verified. Each assembled instruction takes the form of a bit-field and a list of operands, where each operand is a bit-field and has a position.

B. Consistency Check

The next step is to check the consistency of the encoding of instructions. Each instruction's bit-field (in terms of 0's, 1's and x's) comes from the top-level encoding and the operands in the instruction. To check consistency, all of the actual instructions are logically compared to ensure that their bit-field sets are disjoint. The complexity of this check is *O(B*I^2)* where *B* is the number of bits per instruction and *I* is the number of actual instructions present.

C. Header File Generation

After the consistency check the interpreter creates the data structures to hold the different instruction types and their respective operands, as well as mask and signature constants for recognizing the particular instructions. It creates data structures for both big-endian and little-endian hosts.

We follow the style of the disassembler included with the SWARM [5] emulator for the ARM7 processor because of its regularity, clarity and efficiency. The data structure has fields of particular bit widths that correspond to the operands. Data for constants and unspecified portions is replaced with dummy-bit fields. After these are constructed, bit-field mask and signature constants are created to distinguish one instruction type from another. For masks: zeroes and ones in the bit field are mapped to ones in the mask, and don't-cares are mapped to zeroes in the mask. For the signature: zeroes and don't-cares are mapped to zeroes, and ones are mapped to ones. Upon execution, the instruction word is logically AND-ed with the instruction mask, and is of the given type

if the result matches the signature. Figure 10 shows the data structure, mask, and signature generated for the MAC instruction. Additionally, a union containing all of the instruction structures, and a dummy structure that represents the un-decoded instruction word is generated. Figure 11 shows the union for the example ISA.

```
//bitfield=110011xxxxxxxxxxxxxxxxxxxxxxxxxx
typedef struct _MAC_struct {
        unsigned dummy0 : 2;
        unsigned BeginConstants : 4; // Pos:29
        unsigned Rm : 4; // Pos:25
        unsigned Rn : 4; // Pos:21
        unsigned Rmac : 4; // Pos:17
        unsigned dummy1 : 10;
        unsigned Rd : 4; // Pos:3
} MAC_struct;
#define MAC_MASK 0xfc000000
#define MAC_SIG  0xcc000000
```

Figure 10: MAC Structure, Mask, and Signature

```
// the union of ALL of the
// instruction field structures
typedef union _insts {
        unknowni_struct unknown_inst;
        MAC_struct MAC_inst;
        SUB_struct SUB_inst;
        ADD_struct ADD_inst;
} insts;
```

Figure 11: Union for Decoding Instructions

D. Disassembler and Trace Interfacer Generation

The disassembler prints out all of the operand values for a matched instruction. The trace interfacer is more complicated because it interprets specific operand data. The parsed instructions are turned into trace entries that contain: the instruction type, the read operands, and the write operands of the given instruction. This trace takes the form of a class that can be used to quickly retarget trace-driven micro-architectural models to a new instruction set.

V. RESULTS

To demonstrate the power of our language and tool, we implemented subsets of the MIPS, ARM and PowerPC 32-bit RISC instruction sets. Figure 12 summarizes our results. Because of varying infrastructure overhead, the time for setting up and comparing results with a reference simulator is not included. All runs of the interpreter took a negligible amount of time, so we do not include them.

The MIPS subset was checked with small instruction traces from the SPIM emulator [3]. For the PowerPC ISA we implemented an integer subset without any system calls, or exotic memory instructions. The results were then compared against the Microlib

PowerPC ISS [12]. The ARM ISA implementation was verified against SWARM's disassembler. Because of the ARM ISA's compact and logical instruction grouping, we were able to implement an abstracted subset of the major instructions[5]. In addition, the model specifies the illegal instructions in the instruction set that overlap the actual instructions.

	MIPS Integer Subset	**PowerPC Integer Subset**	**ARM (approx.)**
Base Instructions	11	11	6
Actual Instructions	55	91	26
Illegal Instructions	0	0	5
Hours to Enter (appox.)	8	6	6
Header File (# lines)	1370	2424	1026

Figure 12: Results Summary

Creation of the models was greatly aided by constraint checking and by dynamic checking in the decoder-generator software. Many errors such as overlapping encodings, duplicate instruction types, and incorrectly sized bit fields were detected automatically by the language and the interpreter. Furthermore, we ended up extensively pasting subtypes and instances along with parameterization to replicate similarly structured instructions rapidly. We found that using collapse instructions and external connections to instruction anchors to be much slower than pasting and parameterization.

VI. FINAL WORDS

With ISA_ML we demonstrated the power of applying domain-specific modeling to the domain of instruction set modeling. By applying meta-modeling we were able to construct a flexible, powerful, extendable and intuitive visual language that is inherently re-targetable. To accompany this language, we constructed an interpreter that checks language rules and then generates C++ code for a disassembler and for a trace-interface for interacting with micro-architectural models.

A. Future Work

Much work remains to be done with both the language and the interpreter. To fully capture complicated instruction sets the language should be extended to allow implicit operands and operands that relate to both memory and registers. We will modify our interpreter code to allow for arbitrarily large instruction sizes (currently only instructions 32-bits or larger are supported). Another interesting direction would be to allow the chaining of multiple instructions within a single instruction to support VLIW-style instruction sets. While this language does allow rapid description of instruction sets, there still is too much redundant information and the interface could be made more intuitive. The interpreter still is not general enough, and the generated code can be further optimized. Other potential directions include: creating interpreters to export

[5] These instructions include: multiplies, non-multiply data-path instructions, the branch instruction, coprocessor instructions, load instructions, and store instructions, and the software interrupt instruction.

models to architectural and hardware description languages, supporting variable length instructions, and extending the language to support execution semantics.

ACKNOWLEDGEMENTS

This research was funded in part by Semiconductor Research Corporation grant # 837.001 and the MARCO GSRC program.

REFERENCES

[1] Ledeczi A., Maroti M., Bakay A., Karsai G., Garrett J., Thomason IV C., Nordstrom G., Sprinkle J., Volgyesi P, "The Generic Modeling Environment", Workshop on Intelligent Signal Processing, Budapest, Hungary, May 17, 2001.

[2] J. Sprinkle, "Model-Integrated Computing", IEEE Potentials, Vol. 23, No. 1, February/March 2004.

[3] Larus, J., "SPIM: A MIPS R2000/R3000 Simulator" available at: http://www.cs.wisc.edu/~larus/spim.html

[4] F. Balarin, Y. Watanabe, H. Hsieh, L. Lavagno, C. Passerone, A. Sangiovanni-Vincentelli, "Metropolis: An Integrated Electronic System Design Environment", Computer Magazine, April 2003, p. 45-52

[5] Michael Dales – "The SoftWare ARM" available at: http://www.cl.cam.ac.uk/users/mwd24/phd/swarm.html

[6] Rajat Moona, ``Processor Models for Retargetable Tools", Proceedings eleventh IEEE International Workshop on Rapid Systems Prototyping, 21-23 June 2000, Paris, pp. 34-39

[7] S. Pees, A. Hoffmann, V. Zivojnovic, and H. Meyer, "LISA – Machine Description Language for Cycle-accurate Models of Programmable DSP Architectures," Proceedings 1999. Design Automation Conference, pp.933-938.

[8] A. Halambi, P. Grun, V. Ganesh, A. Khare, N. Dutt, and A. Nicolau, "EXPRESSION: A language for architecture exploration through compiler/simulator retargetability," in *Proceedings of the European Conference on Design, Automation and Test (DATE)*, March 1999.

[9] OMG UML Documentation website: http://www.omg.org/technology/uml

[10] D. Burger, and T.M. Austin, "The SimpleScalar Toolset Version 2.0," Tech Report. 97-1342, Department of Computer Science, University of Wisconsin-Madison, June 1997.

[11] Meyerowitz, T., Sangiovanni-Vincentelli, A. "Modeling the XScale and Stongarm processors in metropolis using YAPI", SRC Deliverables Report, April 30th, 2003.

[12] Microlib PowerPC 750 simulator available at: htpp://www.microlib.org