Harmless, a Hardware Architecture Description Language Dedicated to Real-Time Embedded System Simulation

Rola Kassem, Mikaël Briday, Jean-Luc Béchennec, Guillaume Savaton, Yvon Trinquet

Harmless, a Hardware Architecture Description Language Dedicated to Real-Time Embedded System Simulation

Rola Kassem[c], Mikaël Briday[a], Jean-Luc Béchennec[a], Guillaume Savaton[b], Yvon Trinquet[a]

[a] *LUNAM Université, IRCCyN, 1 rue de la Noë, BP 92101 44321 Nantes cedex 3, France*
[b] *ESEO, 4 rue Merlet de la Boulaye, BP 30926 49009 Angers cedex 01, France*
[c] *Beirut Arab University, P.O.Box 11 - 50 - 20 Riad El Solh 11072809 - Beirut, Lebanon*

Abstract

Validation and Verification of embedded systems through simulation can be conducted at many levels, from the simulation of a high-level application model to the simulation of the actual binary code using an accurate model of the processor. However, for real-time applications, the simulated execution time must be as close as possible to the execution time on the actual platform and in this case the latter gives the closest results. The main drawback of the simulation of application's software using an accurate model of the processor resides in the development of a handwritten simulator which is a difficult and tedious task. This paper presents Harmless[1], a hardware Architecture Description Language (ADL) that mainly targets real-time embedded systems. Harmless is dedicated to the generation of simulator of the hardware platform to develop and test real-time embedded applications. Compared to existing ADLs, Harmless 1) offers a more flexible description of the Instruction Set Architecture (ISA) 2) allows to describe the microarchitecture independently of the ISA to ease its reuse and 3) compares favorably to simulators generated by the existing ADLs toolsets.

Keywords: Hardware Architecture Description Language, Instruction Set

[1] This work was supported by ANR (French National Research Agency) under Grant Number ADEME-05-03-C0121.

1. Introduction

The benefits of simulation techniques for the design of software, in particular for embedded systems, do not need to be justified anymore. They are complementary of other V&V techniques (*Verification* & *Validation*), particularly formal V&V techniques often based on coarse-grain models. Another benefit of simulation is the possibility to design and validate software when the hardware is not yet available. By using simulation, both software and hardware can be designed simultaneously and time-to-market is reduced.

Simulation can be conducted at many levels, from the simulation of a high-level model of the application to the simulation of the actual binary code on a time-accurate model of the processor.

In our application field – real-time embedded systems – the simulation of the actual binary code allows to get the closest results from the execution on the actual platform, especially for the execution time. This application field includes – embedded systems –, where executions timings are not a primary scope (functional simulation is sufficient).

Two simulation approaches are attractive. The first approach consists of an Instruction Set Simulator (*ISS*) that takes only the instruction behavior into account independently of the timing of the instruction. The second approach is based on a Cycle Accurate Simulator (*CAS*) which takes into account the instruction behavior and the timing of the real system (it models the internal architecture). Both simulation schemes are interesting. A CAS is slow but essential for real-time system simulation. An ISS has better performances and can be used to quickly execute uninteresting portions of the code (from a real-time system simulation point of view) until the CAS takes over to focus on the interesting parts. Some ISS can

also be associated to a structural simulator to offer both ISS and CAS advantages.

However, the development of a simulator by hand is a difficult and time-consuming process, especially the simulator validation when it targets a complex and modern processor. Moreover, most of this work is not reusable for a new target architecture. So, in order to simplify this task, a Hardware Architecture Description Language (ADL) can be used to describe the instruction set and the microarchitecture of the processor. From this description, a simulator, among other tools, can be generated.

Traditionally, the main goal of an ADL is design space exploration. Such a language allows to describe the instruction set of a processor and the microarchitecture that implements this instruction set. Many ADLs exist such as nML, MIMOLA, LISA, EXPRESSION, ArchC and MADL. These languages are presented in section 2.

This work presents Harmless (Hardware ARchitecture Modeling Language for Embedded Software Simulation), a new ADL. Unlike existing ADLs, it is not dedicated to design space exploration but to the simulation of real-time applications. The goal is to quickly build a simulator to develop and test embedded real-time software, as in Béchennec et al. (2011) for instance. Other differences exist and are highlighted hereafter. Harmless aims at fulfilling the following requirements:

- An incremental and flexible description of the ISA. The language should allow a partial description of the instruction set. For instance, the binary formats of the instructions are needed to generate a decoder but the behaviors are not. So binary formats and behaviors should be described separately. The language should also have the ability to model an ISA with variable length instructions;

- An independent description of the microarchitecture. This point is an important one for 2 reasons: 1) the microarchitecture description is not necessary

for functional simulation (ISS) and may be omitted initially. 2) the same instruction set is shared by many different microarchitectures among a processor family; an independent description allows to share the instruction set description similarly;

- A concise description. The language should be focussed on the operations used by the instructions (bit operations, bit field extraction, *etc.*) and should allow the sharing of sub-descriptions;

- An easy to check description. The language should encompass the whole description (for instance relying on C or on another general purpose language to do the algorithmic parts should be avoided. Indeed it would lead to unchecked description and likely to an erroneous behavior of the simulator). This way the model may be checked extensively and description errors are minimized;

- Good runtime simulation performances. It should compare favorably against other generated simulators.

How Harmless satisfies these requirements is presented in section 3.

In the remaining of the paper, section 4 provides an in-depth presentation of the language, section 5 presents the underlying model of the pipeline, sections 6 and 7 explain how the simulators (ISS and CAS) are generated and show the performances obtained for some processor descriptions (*e.g.*, description length, speed, accuracy).

2. Related work

The goal of a Hardware Architecture Description Language (*ADL*) is the formal description of a hardware architecture. An ADL differs from a Hardware

Description Language (*HDL*) by the abstraction level of the description. Well-known HDLs such as *Verilog* or *VHDL* focus on the register transfer level and do not provide a straightforward method to model instruction sets. An ADL allows to model a processor architecture and aims at the automatic generation of tools such as simulators, compilers, assemblers and linkers. First, this section will examine a classification of ADLs. Then it will present some existing ADLs.

2.1. Classification

In this section we classify ADLs in the context of embedded processor design (Mishra and Dutt (2008)), taking into consideration either *Contents* or *Objectives*, as shown in figure 1.

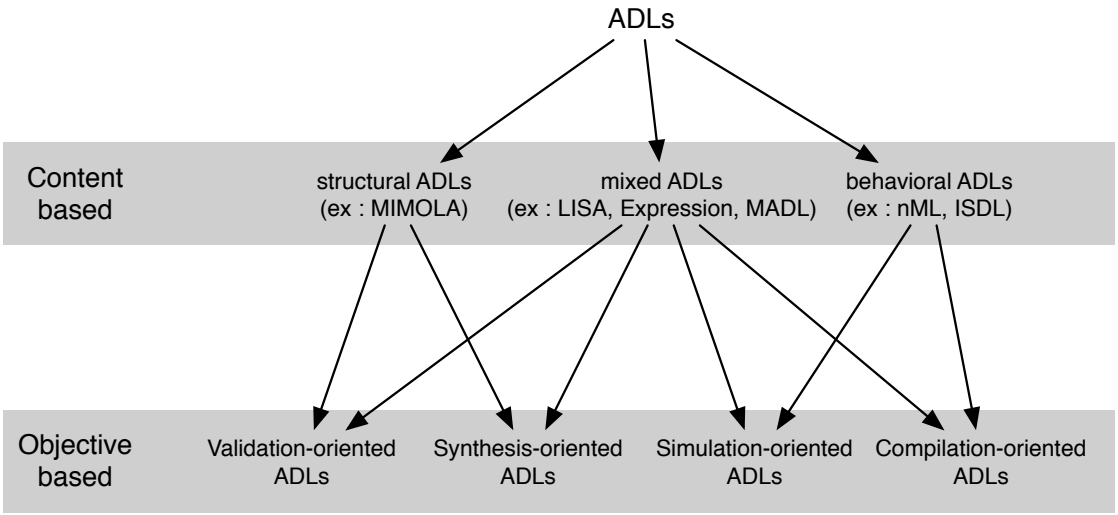


Figure 1: ADLs classification based on contents and objectives (Mishra and Dutt (2008))

According to the *contents* of the description, ADLs may be classified in three categories:

- *behavioral*: The ADL focuses on the instruction set and does not model structural details. This type of ADLs can be used to generate an Instruction

Set Simulator (*ISS*). *nML* (Freericks (1991); Fauth et al. (1995)) and *ISDL* (Hadjiyiannis et al. (1997)) are some examples;

- *structural*: The ADL models structural aspects of the processor (typically RT-level). It can be used for cycle accurate simulation, but does not really model the instruction set. *MIMOLA* (Bashford et al. (1994); Zimmermann (1979)) is an example;

- *mixed*: The ADL captures both the structural and behavioral aspects. It combines the benefits of the two other types. *LISA* (Zivojnovic et al. (1996); Hoffmann et al. (2002)), *EXPRESSION* (Halambi et al. (1999)) and *MADL* (Qin (2004)) are some examples.

According to the *objectives*, ADLs may be classified in four categories:

- *synthesis*: ADLs with a detailed model of the structural aspects of the processor are suitable for hardware generation;

- *validation*: This kind of ADLs is suitable for functional verification of embedded processors against processor specification;

- *compilation*: This kind of ADLs can be used to generate retargetable compilers;

- *simulation*: ADLs that can generate a simulator of the processor. Behavioral simulators are limited to instruction set simulators, whereas mixed simulators can generate both ISS and CAS.

In reference to this classification the goal of Harmless is to generate both ISS and CAS simulators. So Harmless is a mixed ADL from the content point of view and a simulation oriented ADL from the objective point of view.

2.2. Behavioral ADLs

Behavioral ADLs aim at the description of the behavior of the processor without detailing the hardware components. Two behavioral ADLs are briefly described hereafter.

2.2.1. nML

The nML language (Freericks (1991); Fauth et al. (1995)) was designed at the Berlin Institute of Technology in Germany. The instruction set of the processor is described as an attributed grammar (Paakki (1995)) that allows a hierarchization and a sharing of common sub-parts of the description. This way, the description of an instruction is viewed as an *and-or* tree where an *or* node expresses the alternative common parts of the instruction and an *and* node lists the composite parts (their operands for instance). This hierarchization leads to a concise description. Each node bears attributes like the *image* (binary format), the *syntax* (textual representation) and the *behavior*. From the description, a functional simulator can be generated.

nML is a simple and easy to learn language. However, variable-size instruction sets are hard to describe and the *image* attribute does not support operations on bit fields to decode and disassemble the instructions needing them.

nML extensions exist, such as Sim-nML (Rajesh and Moona (1999)) and GLISS-nML (Ratsiambahotra et al. (2009)). Moreover, it has been used by the instruction set simulator Sigh/Sim (Lohr et al. (1993)), the code generator CBC (Fauth and Knoll (1993)) and the instruction simulation environment CHESS/CHECKERS (Target (2003)).

2.2.2. ISDL

The Instruction Set Description Language *ISDL* (Hadjiyiannis et al. (1997, 2000)) has been developed at the Massachusetts Institute of Technology (*MIT*),

Cambridge, USA. Like nML, ISDL allows to describe the instruction set of the processor as an attributed grammar. It has been used by the compiler AVIV (Hanono and Devadas (1998)) and the simulator generator Gensim (Hadjiyiannis et al. (1999)).

ISDL is more flexible and the semantics are stronger than that of nML. It allows the description of a wide variety of architectures, with emphasis on *VLIW* architectures (*Very Long Instruction Word*). Like in nML, the instruction set description contains both structural and behavioral information. In addition, it allows the description of the hardware configuration of the microarchitecture which is mixed with the instruction behaviors.

An ISDL description is composed of six parts:

- The *Instruction Word Format* part to describe the architecture word instruction;
- The *Storage Resources* part to describe the size of memory, register files, and special registers;
- The *Global Definitions* part to define tokens (can be used to group syntactically related entities such as register names), non-terminals (allow the sharing of common structures in the operation definitions, eg, addressing modes), and split functions (define how a long constant, for example a long memory address or immediate data, can be divided into several subfields in the binary word of the instructions) that are used in other parts of the description;
- The *Instruction Set* part to define the various operations using the functional units, memories and buses;
- The *Constraints* part uses a set of boolean rules for the compiler to define

all operations that cannot be executed in parallel;

- The *Optional Architectural Details* part to give optimization information. This is useful for the compiler to optimize the generated code.

ISDL can generate an instruction set simulator, but also a temporal simulator based on the timing information specified in the operations. However, there is no way to model the instructions dependancies and the concurrency found in a pipeline.

As a conclusion, the behavioral ADLs allow a hierarchical description of instruction sets based on an attributed grammar. This feature allows the sharing of common parts between the instructions by grouping them thereby simplifying the description. But it is not possible to generate a cycle accurate simulator due to the lack of some structural details (timing informations and the possibility to describe the real operation of a pipeline).

2.3. Structural ADLs

In this section, we present some aspects of the MIMOLA ADL (Zimmermann (1979); Bashford et al. (1994); Mishra and Dutt (2008)). This language is focused on modeling the internal structure of a processor and targets hardware synthesis. It was developed by Gerhard Zimmermann and a group of researchers at Radio Astronomy Observatory of the University of Kiel, Germany.

MIMOLA is one of the first languages specifically designed for high-level synthesis of processors. It allows to describe the hardware structure in the form of a netlist of interconnected *modules*. In a MIMOLA description, the behavior of a module is written in the *program* section by using a language similar to Pascal. The user may choose to write the behavior at the application level or write an instruction interpreter. The latter enables to obtain an instruction set in the final design because the synthesis will generate a microcoded interpreter.

In general, MIMOLA is seen as a very low-level language and descriptions are laborious to write and modify. As a consequence, the generated cycle accurate simulators are slow.

2.4. Mixed ADLs

This section focuses on LISA, a mature mixed ADL that is now an industrial product of *Synopsys*, EXPRESSION which is targeted at design space exploration of System on Chip (SoC) architectures, MADL, which main idea is the use of the Operation State Machine (OSM) to model microprocessors and ArchC, a SystemC framework.

2.4.1. LISA

LISA was first introduced in Zivojnovic et al. (1996) and presented as a machine description language that gives a formal description of programmable architectures, their interfaces and peripherals. It allows the automatic (or semi-automatic) generation of many tools such as C compiler, assembler, linker, simulator, profiler and VHDL code generator for synthesis.

in Pees et al. (1999), six models of a LISA description are presented:

- the *memory model* describes storage parts, including registers and main memory (size, alignment, *etc.*);
- the *resource model* describes hardware resources and their access properties (read/write capability for registers, for instance);
- the *behavioral model* describes the hardware activity. The basic concept is to represent the system behavior as a state machine;
- the *instruction set model* collects the hardware operations related to instructions, and verify their compatibility against the actual hardware;

- the *temporal model* defines sequences between instructions, including waiting states;

- the *microarchitecture model* is used to group functionalities into one entity (i.e. both addition and subtraction are part of the ALU).

The simulator generation requires the description of memory (simulation of storage), behavior (operation simulation), instruction set (decoder/disassembler) and timing models (operation scheduling).

In many aspects, the Instruction Set Architecture (*ISA*) defined in LISA includes ideas which are analogous to those of nML, and the binary format, behavior and syntax of instructions are placed in the same view.

LISA allows to describe a pipeline in an abstract way (*i.e.*, the designer does not need to give the structure of the processor), but pipeline registers are explicitly defined and instructions should reference these registers in their behavior. As a result, when using the same ISA for different microarchitectures (case of PowerPC or ARM instruction sets for example) the designer is forced to rewrite the behavior of instructions.

2.4.2. EXPRESSION

EXPRESSION (Halambi et al. (1999)) is a mixed ADL for modeling, design space exploration and verification of SoC which has been developed at the University of California, Irvine. From the descriptions, tools like simulator and compiler may be generated.

A description is composed of two sections: the *structure* and the *behavior*. The structure section has the following subsections:

- the *components description*: components are buses, functional units, *etc.* A component may have attributes like ports, connections, accepted opcodes and so on;

- the *pipeline and data-tranfer paths description* describes the arrangement of the components and the connections between them. The pipeline describes the instruction flow and the available data paths;
- the *memory subsystem* describes the memory subsystems components and their connectivity.

The behavior section has the following subsections:

- the *operation specification* describes the instruction set of the processor. Instructions having common characteristics are gathered in *opgroups*. For each instruction, its operands, binary format and behavior are described.
- the *instruction description* exhibits the parallelism within instructions to support VLIW instruction sets;
- the *operation mapping* gives the informations needed by a compiler for optimization purpose.

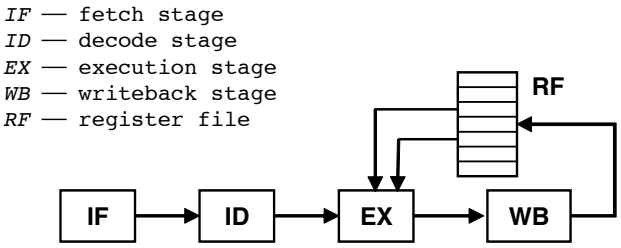
SIMPRESS, the simulator generator, takes an EXPRESSION description as input and generates an interpretative cycle accurate simulator. Authors reported that the simulation is slow because of interpretative simulation. Another reason would be that the boxes and wires model used in EXPRESSION incurs additional data copy and synchronizations.

2.4.3. MADL

MADL (Qin (2004); Mishra and Dutt (2008)) is an Architecture Description Language designed at University of Princeton (USA) that primarily targets cycle accurate simulators and instruction set simulators. Its main characteristic is to associate an Operation State Machine (OSM) to each instruction. As in figure 2,

extracted from Qin et al. (2004), the `ADD` instruction is modeled using a 5 states OSM and the pipeline model is composed of 4 stages.

In the description, a section `MACHINE` describes the authorized states and edges in the microarchitecture, which gets the instruction flow in the pipeline (using keywords dedicated to automata description such as `INITIAL`, `STATE`, `EDGE`).



(a) A 4-stage pipeline

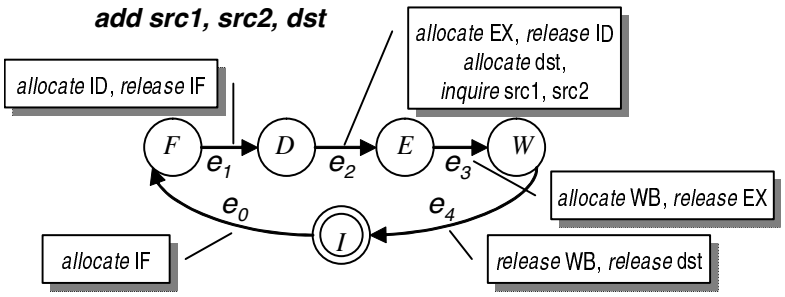


Figure 2: Madl OSM model example of the ADD instruction (Qin et al. (2004)).

Instruction set description is largely inspired by nML and LISA and uses an AND-OR graph, which is a directed acyclic graph with only one source node. There is one graph to describe both binary coding, syntax and behavior. However, the instruction behavior description based on OSM is largely different from nML and LISA because of the underlying OSM model. The description hooks the behavior of the instruction onto edges of the OSM. For instance, in figure 2, the edge from states E to W (called e_ex_w) may be described:

```
TRANS
```

```
e_ex_w: v_rd = v_rn + v_oprnd2;
```

The other parts of the description (reading operands, writing back result) are described in less specific nodes dedicated to all triadic instructions.

The main advantage of this approach is the fast generated simulators (both CAS and ISS). However, since each instruction behavior uses an OSM, the ISA is tied to a particular microarchitecture and cannot be easily retargeted to a different microarchitecture.

2.4.4. ArchC

ArchC (Azevedo et al. (2005)) is based on *SystemC* (Kranen (2006)), a set of *C++* libraries for electronic system modeling. As such, *ArchC* is not a self-contained language, but a set of classes, data types and macros that can be compiled using the *ArchC* preprocessor and a standard-compliant *C++* compiler.

From an *ArchC* model, several kinds of simulators can be generated: in *interpreted* simulators, the source model is transformed into *passive C++* objects that will be processed at runtime by a common simulation core; in *compiled* simulators, the hardware architecture is mimicked by a set of custom *C++* classes generated from the source model. *ArchC* can be used to generate instruction set simulators as well as cycle accurate simulators.

In *ArchC*, a processor model is composed of two sections:

- A description of the architectural resources: registers are declared and tied to *register formats*. Additional resources include internal memory (for microcontrollers), communication ports and pipeline stages.

- The instruction set: all instructions are declared separately and tied to *instruction formats*. This section also details the assembly language syntax, the decoding process, and the duration (in clock cycles) of each instruction.

The behavior of the architecture is described as a set of *C++* blocks attached either to specific instructions, to specific instruction formats, or to all instructions. In each of these blocks, the recommended style is to use *switch* statements to decide which operations will be performed depending on the current pipeline stage. Apart from standard *C++* constructs, *ArchC* provides facilities for describing register transfers, memory accesses and pipeline control (*stall*, *delay*, *flush*).

Compared to other ADLs presented in this section, *ArchC* appears more as a set of facilities for writing simulators in *C++*, rather than a domain-specific language for hardware architecture modeling. While *nML* was based on a grammar-based approach for modeling common aspects of instructions, *ArchC* provides non-hierarchical *instruction formats* and requires a separate description of the assembly language and decoding for each instruction.

Finally, in *ArchC*, the functional behavior of instructions (arithmetic operations, register transfers) is mixed with the description of the pipeline control. As a consequence, it will be more difficult to check the consistency of the model, to make sure that modifications in one aspect do not impact the others, or to provide different pipelines for the same instruction set.

3. Harmless requirements

As we have seen in section 2, most of the existing ADLs use a hierarchical description of the ISA and allow to share common parts between the instructions. However each aspect of the description (the binary format, the textual representation and the behavior) are put in the same hierarchy. This leads to a lack of flexibility because, in most ISA, the best hierarchical description is not the same for each aspect.

Mixed ADLs are needed for cycle accurate simulation. However, the pipeline

description is either mixed with the instructions behavior (LISA, MADL) or is done using an explicit component net list, pipeline paths and data-transfer paths (EXPRESSION). As a result, changing the pipeline description is difficult whereas it is a common task in the development process in order to investigate various micro controllers in the same family.

As shown in this section, Harmless addresses those issues by splitting the ISA description in 3 views and by mapping *devices ports* used in the behavior view onto the pipeline. Variable length ISA are supported too.

3.1. A simulation oriented mixed ADL

The main goal of Harmless is to provide support for both ISS and CAS simulator generation. Designers will use the generated simulators to verify embedded systems (both functionally and temporally); so Harmless does not focus on Design Space Exploration: It is not designed to refine the description to allow hardware synthesis (as *VHDL* or *Verilog* languages for instance). This difference is fundamental because the underlying internal model does not need to reflect the exact structural hardware characteristics. In reference to figure 1, Harmless is a "mixed ADL" for the content based criterion and "simulation oriented" for the objective criterion.

The main expected characteristics are listed in section 1. The main features of Harmless are defined out of these specifications and significantly differ from other ADLs.

3.2. Incremental and flexible description of the ISA

Unlike other ADLs, Harmless provides a separate view for each aspect of an ISA description (binary format, behavior and syntax). A set of trees composes each of these views and each may have a different structure. This feature enables the designer to choose the best structure for each view.

This approach allows to do the description of an ISA in an incremental and flexible way. For example, ISA have often optional instructions that may be available on some micro-controllers only. In a Harmless description, the binary format view would describe the whole ISA binary format. For the optional instructions, the behavior view would either trigger an *instruction unavailable* exception or provide the behavior of the optional instructions. The ISA description in Harmless is detailed in sections 4.1, 4.2 and 4.3.

Harmless allows to describe variable-length instruction sets. This feature is handled in the format view by adding more bytes as a specific format is described. Variable-length ISA description is detailed in 4.1.1.

3.3. Independent description of the microarchitecture

Information needed to compute the timing of the instructions are not part of the ISA description. They are provided in the microarchitecture view through a mapping of *hardware devices ports* (ALU operations, memory read/write, *etc.*) used in the behavior view of the ISA onto the *pipeline*.

That way, the same ISA (ARM for example) may be mapped on different microarchitectures easily (*e.g.*, ARM7, ARM9 and ARM11). Unlike the way it is managed in other ADLs, this description is very concise and easy to maintain. This mapping and an example of the retargeting of the PowerPC ISA behavior are shown in section 4.4.

3.4. A concise description

Harmless differs from a general purpose language (*e.g.*, C, C++) by data manipulation at the bit level. Variables are defined with their actual number of bits and bit field extraction and concatenation are supported with a simple syntax. In each view, the description can be split to multiple sub-descriptions to

share common parts. Input and output arguments and components' methods may be used in the behavior view.

3.5. An easy to check description

Harmless is a very easy to learn imperative language. It is a strongly typed language. Arithmetic and shift operations do not implicitly overflow. For instance, adding two n bits variables produces a $n+1$ bits result that requires a $n+1$ bits variable to be stored.

No external C functions are allowed (contrary to LISA), so that semantic verifications remain possible at compile-time. For example, the Harmless compiler checks that: 1) two different instructions may not share the same binary code which is an ambiguous description (see section 6.2); 2) the mapping of instruction onto the pipeline is coherent (see section 7.1.1).

The parser is written using an unambiguous LL1 grammar.

3.6. Good runtime simulation performances

One of the most penalizing architectural construct to simulate, in terms of computation, is the pipeline. Because of structural, data or control hazards, its behavior needs to be modeled to get a cycle accurate simulation. A classical way is to implement the pipeline using a register transfer level (RTL) model. This is a typical approach used by HDLs or by SystemC (Panda (2001)). As explained in the beginning of this section, our language does not target hardware synthesis and thus may use other internal representations as long as functional and temporal behaviors of the simulator are accurate. In Harmless, we use an internal model for the pipeline based on finite automata where each state represents a state of the pipeline at a given time, and one transition is taken at each clock cycle. The objective with this model is to bring a substantial part of the computation time required to simulate the pipeline in the generation of this automaton, at compile

time. Thus, the runtime overhead is lower than when using a classical approach. This approach is detailed in section 5.

A cycle accurate description of the memory hierarchy (memory cache, memory latency, *etc.*) is not presented in this paper and is still a work in progress.

Performance comparison with other simulators, both handwritten and generated are made in section 7.2

4. Description of the language

Harmless uses 3 views to describe the ISA of the processor (as presented in Kassem et al. (2009b)):

- the *format* view deals with the binary format of the instructions;
- the *syntax* view describes the textual format of the instructions;
- the *behavior* view describes the behavior of the instructions.

Each view is a set of trees where a node describes a piece of *format*, *behavior* or *syntax* (*i.e.* the kind of the node). The temporal behavior is not part of the ISA.

The goal of each description is to share most of the common properties of each view. So, like in a grammar specification language, each view allows to describe whether a non-terminal node is built by aggregating sub-nodes (common properties like addressing modes) or by selecting one node among several. By default, a non-terminal node aggregates the sub-nodes. The *select* instruction allows to choose one sub-node among several (or an aggregate of sub-nodes among several).

Each instruction is represented as a branch in a tree. Instructions sharing a common part in a view share nodes in the roots of the tree, while specific parts are located in leaves.

A node may have one or more *tags*. A set of tags along a branch of a tree is the unique identifier of an instruction and is called the *signature of the instruction*.

Tags appear in the *description* part of a node. The syntactical representation of a tag starts with the '#' character, followed by an identifier. The signature of the instruction is used to link instructions over the different views.

Let's consider the `ADD` instruction of the PowerPC ISA as a guiding example in the following sections. `ADD` stores the sum of registers `rA` and `rB` into register `rD`. It behaves differently if the overflow is computed (suffix `o`) or if the condition register is updated (suffix '`.`'). See figure 3.

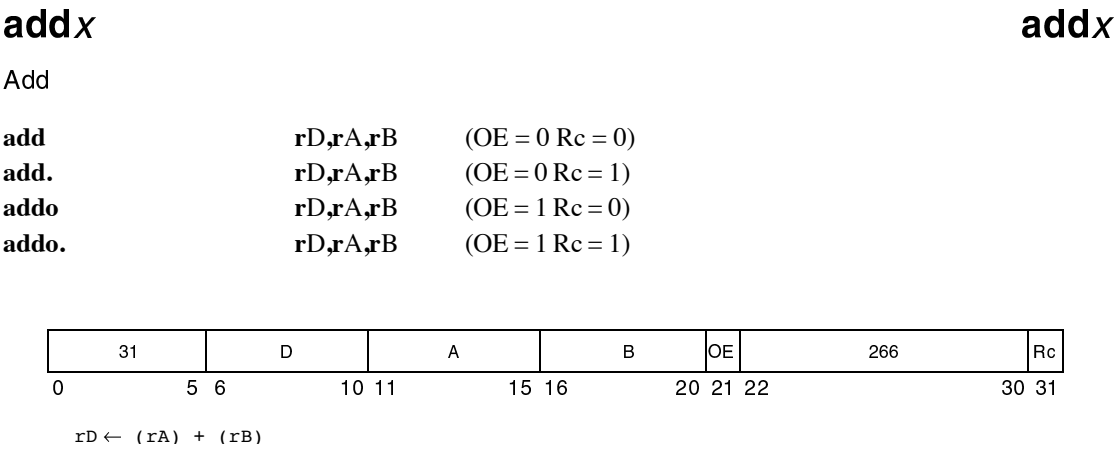


Figure 3: Representation of the `add` instruction in the PowerPC instruction set (Freescale (2005)). `D`, `A` and `B` refers to register indexes. Combinations of the `Rc` and `OE` fields give 4 instruction variants.

4.1. The format view

The format view describes the binary format of the instructions. It extracts both the opcode and fields that are used in other views (behavior and syntax). By default, a node aggregates the different declarations. The format tree representation of the `ADD` instruction is presented in figure 4.

The root node (`Instruction`) has to distinguish the *add* instruction in the whole instruction set, based on a part of the binary code, in this case, the 6 most

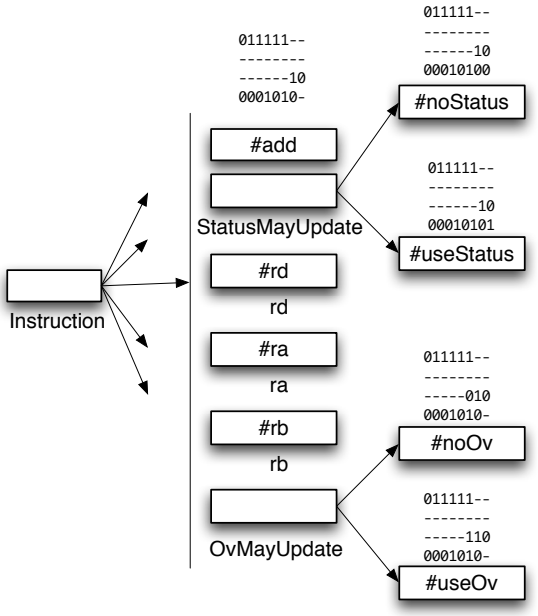


Figure 4: Tree representation of the `ADD` instruction format and its 4 variants. Binary numbers on top of nodes represent the binary format of instruction part. Node names are given under nodes, if this is appropriate.

significant bits (bits 31 to 26) and the 11 less significant bits (bits 10 to 0). Note that contrary to the PowerPC instruction set, Harmless considers bit 0 as the least significant bit. So bits 31 to 26 in the Harmless description are bits 0 to 5 in figure 3 and bits 10 to 0 are bits 21 to 31.

The root node description would be:

```
format Instruction -- decode opcode
  select slice{31..26,10..0}
    case \m011111_-10_0001_010- is
      inst_add
    case ...
  end select
end format
```

The *select* structure uses a bit mask (a binary number prefixed by `\m`) to indicate which part of the slice is meaningless for the differentiation of the instructions: bits denoted with the '-' are irrelevant for the comparison. The underscore ('_') is not taken into account and is used only to ease the readability. In that description, if an instruction has the 6 most significant bits (bits 31 to 26) as `011111` and the 11 less significant bits as `-100001010-`, then this is an `ADD` instruction: the format node `inst_add` is referenced.

The `inst_add` would be:

```
format inst_add
  #add
  StatusMayUpdate
  rd
  ra
  rb
  OvMayUpdate
```

```
8 end format
```

This node is the aggregation of:

- a *tag*, that will be used to identify an instruction (through its signature): `#add`;
- calls to different other sub-nodes that are aggregated.

The `StatusMayUpdate` sub-node differentiates instructions that update the condition code register (CCR) from others. The bit 0 (field `Rc`) is used here:

```
1 format StatusMayUpdate
2   select slice{0}
3     case 0 is #noStatus
4     case 1 is #useStatus
5   end select
6 end format
```

The `OvMayUpdate` sub-node differentiates instructions that update the overflow flag, based on bit 10 (field `OE`). Its structure is identical and provides either tag `#noOv` or tag `#useOv`.

Sub-nodes `ra`, `rb` and `rd` refers to field operand extraction. They could be declared directly inside the `inst_add` node, but with this approach, the `rd` node may be called by other nodes (register operands are often at the same place in an instruction set).

```
1 format rd
2   #rd
3   rD := slice{25..21}
4 end format
```

At line 3, a format field is extracted from the binary code of the instruction: `rD` is obtained from the 5 bits (25 to 21). A slice may be signed or unsigned and has an implicit type that depends on the number of bits it uses. In the example above, `rD` is an unsigned 5 bits integer (*u5* type in Harmless). Some basic operations like left or right shifting and field concatenation may be performed when a field is extracted.

Eventually, the description of the `add` instruction leads to generate 4 branches in the description tree (see figure 5), with the definition of 4 instructions in Harmless.

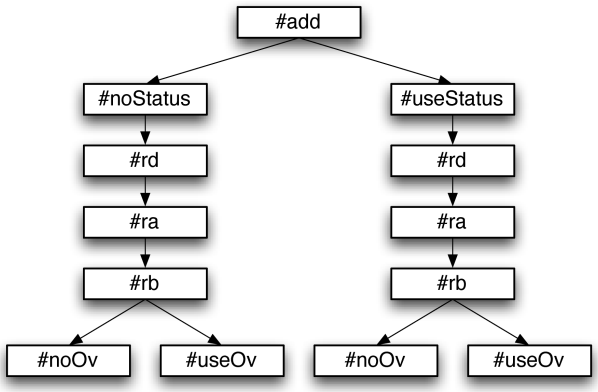


Figure 5: Graphical representation of the description tree generated for the 4 variants of the add instruction.

4.1.1. Variable-length instructions

Harmless format description supports variable-length instructions. When an instruction is lengthened by adding a slice, the new length is valid for all the children nodes but not for the sibling or parent nodes which use the previous instruction length. The incremental instruction description is flexible: an addressing mode that requires to decode an extra word is described only once, even if other instructions use it in another place (there was another word added earlier in the de-

scription tree). Here is an example extracted from the Atmel AVR instruction set. Instruction binary code uses 16 bits on most instructions, except the `CALL` and `JMP` that requires an extra word, as indicated by the '+' in `slice {15..0}+{15..0}` line 3 of the following description. The `addr` operand extracts 24 bits from this instruction line 5.

```
1  -- opcode is:
2  -- 1001 010K KKKK 11-K KKKK KKKK KKKK KKKK
3  format longCall slice {15..0}+{15..0}
4    #LONGCALL
5    addr := slice{8..4,0}{15..0}
6    select slice {15..9,3..1}{-}
7      case \b1001_010_111 is #CALL
8      case \b1001_010_110 is #JMP
9    end select
10 end format
```

4.1.2. Sub-format reuse

In some occasion, the same sub-format is used in more than one place (the addressing mode is used twice). To differentiate same sub-formats, a tag suffix should be added to the node name. For instance, in the HCS12 description, the indexed addressing mode sub-format (xb_am) may be used in both the source and the destination operands. This is indicated using the following description:

```
1 format idx_idx
2   xb_am@SRC
3   xb_am@DST
4 end format
```

where @SRC and @DST are the suffix (added at the end of sub-formats, to differentiate them).

4.2. The syntax view

The syntax view describes the textual format of the instructions that is suitable for program disassembly. This view is not required to generate only a simulator, however this could help to disassemble programs during the debugging phase[2].

This view binds a textual syntax to each instruction signature. As in the format view, syntax nodes are associated to tags that are part of the signature. The textual representation of the instruction is built by concatenating character strings along the branch of the instruction description.

To highlight syntax description characteristics, we use the same example than in the previous section, with the 4 variants of the `add` instruction. Its signature is: `#add`, either `#noStatus` or `#useStatus`, `#rd`, `#ra`, `#rb` and either `#noOv` or `#useOv`. The root node is:

```
1 syntax addInst
2   #add
3   "add"
4   useOvFlag
5   useStatus
6   " "
7   AMrDrArB
8 end syntax
```

Line 2 refers to a tag. Line 3 and 6 are simple character strings: These are parts

[2]For instance, the comparison of the disassembly generated by Harmless and by the GNU `objdump` disassembler helped to validate both binary and syntax descriptions of PowerPC, ARM and AVR ISA.

of the instruction mnemonic. `useOvFlag` and `useStatus` and `AMrDrArB` refers to sub-nodes.

```
1 syntax useOvFlag
2  select
3   case #noOv
4     -- nothing to write
5   case #useOv
6     "o"
7  end select
8 end syntax
```

As previously explained in the ISA description in figure 3, the `add` instruction that deals with overflow flags is appended with a '`o`' at the end of the mnemonic. the `useStatus` nodes has the same structure and adds a '.' at the end of the mnemonic if the status register (CCR) should be updated. The `AMrDrArB` embeds the syntactical description of all triadic instructions and refers to operand fields decoded in the format part. Fields are typed (`u5` here) and are checked against the format view:

```
1 syntax AMrDrArB #rd #ra #rb
2  field u5 rD
3  field u5 rA
4  field u5 rB
5  "r\d,r\d,r\d", rD, rA, rB
6 end syntax
```

In a syntax node, standard complex control structures such as `if...then...else` can be used. This is needed to give more flexibility to the syntax. For instance, when a field has a special value, the instruction may be viewed as a special one too. This is often the case in RISC instruction sets, such as the PowerPC, where the

`addi rD,0,value` (*add* *i*mmediate) instruction translates to the simplified syntax `li rD,value` (*l*oad *i*mmediate):

```
1  syntax addi
2    #addi #rd #ra #simm #noStatus #noOv
3    field u5 rA
4    field u5 rD
5    field s16 SIMM
6    if rA = 0 then
7      "li r\d,\d",rD,SIMM
8    else
9      "addi r\d,r\d,\d", rD, rA, SIMM
10   end if
11 end syntax
```

4.3. The behavior view

This last view is the most complex one. The behavior view binds a behavior to each instruction signature and provides a way to describe the components which are accessed by instructions. The description of the components is made in an object-oriented way and contains data as well as methods. Methods are used by the instruction set. For instance, a register file component provides read from register and write to register methods; an Arithmetic and Logic Unit component provides add, subtract, bitwise or, bitwise and, xor methods; a memory component provides read and write methods with several data width. However, the description is only functional and the time needed to invoke a method provided by a component or the concurrency allowed to a method are not described at that stage but in the microarchitecture view as shown in section 4.4.

For instance the following description shows a part of the alu component and one of its methods for the PowerPC model:

```
component ALU {
  -- condition register is split in 8 parts
  register u32 CR{
    CR0 := slice{31..28}
    CR1 := slice{27..24}
    CR2 := slice{23..20}
    CR3 := slice{19..16}
    CR4 := slice{15..12}
    CR5 := slice{11..8}
    CR6 := slice{7..4}
    CR7 := slice{3..0}
  }

  -- update CR0 when needed
  void updateStatus(u33 result){
    s32 tmp := (s32)(result{31..0})

    if tmp = 0s then
      CR.CR0 := 2 -- EQ
    elseif tmp > 0s then
      CR.CR0 := 4 -- GT
    else
      CR.CR0 := 8 -- LT
    end if

    CR.CR0{0} := XER.SO -- Integer exception register summary
        overflow bit
  }
```

```
28    -- ...
29 }
```

As for the two other views, the remaining of the behavior view is a set of nodes which describes a piece of behavior. A behavior node contains a declaration section with local variable declarations and other behavior node references, and one or more `do` blocks to specify the algorithm of the instruction.

Harmless is a strongly-typed language. Since it is targeted at instruction set description, it offers signed and unsigned data types with any number of bits. The language has some important features. For instance, the sum of two n-bit words produces an $n + 1$-bit result, thus the result is not truncated. This way, the implementation of the carry or overflow computation is easier. Operators to extract and concatenate bit fields are provided:

```
1  u16 val1 := \x5500
2  u16 val2 := \x0055
3  u17 result := val1+val2  -- 17 bits
4  u1 carry := result{16}   -- only MSB
5  u16 valResult := result{15..0}
```

The following description is the behavior of the PowerPC `ADD` instruction:

```
1 behavior add_inst #rd #ra #rb #add
2   field u5 rD
3   field u5 rA
4   field u5 rB
5   u33 result
6   u32 op1
7   u32 op2
8   do
9     op1 := SRU.GPR.read32(rA) -- System Register Unit General
```

```
        Purpose Register
10      op2 := SRU.GPR.read32(rB)
11      result := ALU.addInt(op1, op2)
12    end do
13    select
14      case #useOv
15        do ALU.updateOverflow(result) end do
16      case #noOv
17    end select
18    select
19      case #useStatus
20        do ALU.updateStatus(result) end do
21      case #noStatus
22    end select
23    do
24      SRU.GPR.write32(rD, result{31..0})
25    end do
26  end behavior
```

Here, the `add_inst` behavior declares 3 local variables to store the registers contents (`op1` and `op2`) and the result of the operation. Then, it performs the addition, using the related component (`ALU.addInt`). Eventually, the 4 variants of the instructions are described (related to the overflow and the status register) and the result is written back. Other behaviors may be called to describe the tree.

4.4. The microarchitecture view

The microarchitecture view describes the microarchitecture (*i.e.* pipeline, hardware constraints) that implements the instruction set of the processor. It maps the instruction set behavior view onto the microarchitecture description using the

components as shown hereafter on figure 6. This work presented in this section is based on work previously published in Kassem et al. (2009a) with some corrections about the pipeline description and extensions about the simulator generation. Moreover, each device is now an instance of a component. Since the instruction set refers to components, it remains independent of a specific microarchitecture while devices allow to specify more than one instance of a component.

The microarchitecture is described in `architecture` and `pipeline` subviews. The `architecture` subview is the interface between a set of hardware components (*e.g.*, registers, memory, ALU) and the definition of the pipeline. It allows to express hardware constraints having consequences on the temporal sequence of the simulator. It may contain many `device`s to control the concurrency between instructions to access the same component.

Every `device` in the `architecture` is related to one component. The methods of a component can be accessed by a `port` that allows to control the competition during access to one or many methods. A port may be private to the microarchitecture or shared (*i.e.* the port is not exclusively used by the microarchitecture, it can be used by other bus masters for instance).

The `pipeline` subview describes a pipeline. A pipeline is mapped onto an `architecture` subview and all stages are listed in order. In each stage, devices and ports that can be accessed by the instruction set (through components' methods) are enumerated.

For instance, let's consider the PowerPC 5516 (e200z1 core, Freescale (2008)). The e200z1 core has 4 pipeline stages:

IFETCH Instruction Fetch From Memory;

DECODE/EA Instruction Decode / Register Read / Operand Forwarding / Effective Address Calculation;

EXECUTE/MEM Instruction Execution / Memory Access;

WB Write Back to Registers

Needed devices and ports are:

- the memory with a *fetch* port for instruction fetch and a *read/write* port for data access (program and data may be accessed simultaneously when they are stored in flash memory and static ram respectively);
- the register file with a *read* port supporting 3 simultaneous reads and a *write* port supporting 2 simultaneous writes;
- the arithmetic and logic unit;
- the effective address calculation unit;
- the branch processing unit.

The microarchitecture of this microprocessor is described through the 2 objects (`architecture` subview and `pipeline` subview). The description of the first object (`architecture`) is as follows:

```
1  architecture PPC5516 {
2    device SRUDev : SRU {
3      read is  GPR.read8    | GPR.read16   |
4               GPR.read32   | spr.read
5      write is GPR.write8   | GPR.write16  |
6               GPR.write32  | spr.write
7      port rs : read (3);
8      port rd : write (2);
9    }
10
```

```
11   device ALUDev : ALU {
12     -- an empty method list means all the methods
13     port all ;
14   }
15
16   device EAUDev : effective_address_Unit {
17     port effAddrUnit : eff_addr_add;
18   }
19
20   device memDev : mem {
21     read is  ram.read8  | ram.read16  |
22              ram.read32
23     write is ram.write8 | ram.write16 |
24              ram.write32
25     shared port fetch : read;
26     shared port loadStore : read or write;
27   }
28
29   device BPUDev : BPU {
30     port branch : absBranch;
31   }
32 }
```

In this description, the devices are declared. For example, the device `memDev` controls the concurrency to access the `mem` component (*i.e.* memory) by two shared ports `fetch` and `loadStore` (*i.e.* this access can be made concurrently by other bus masters). The port `loadStore` allows to access the methods `read` or `write`. This is an exclusive access, *i.e.* if an instruction uses `read` in a stage of the pipeline, the second method `write` may not be used concurrently. `Read` is an alias of `read8`,

`read16` and `read32` methods.

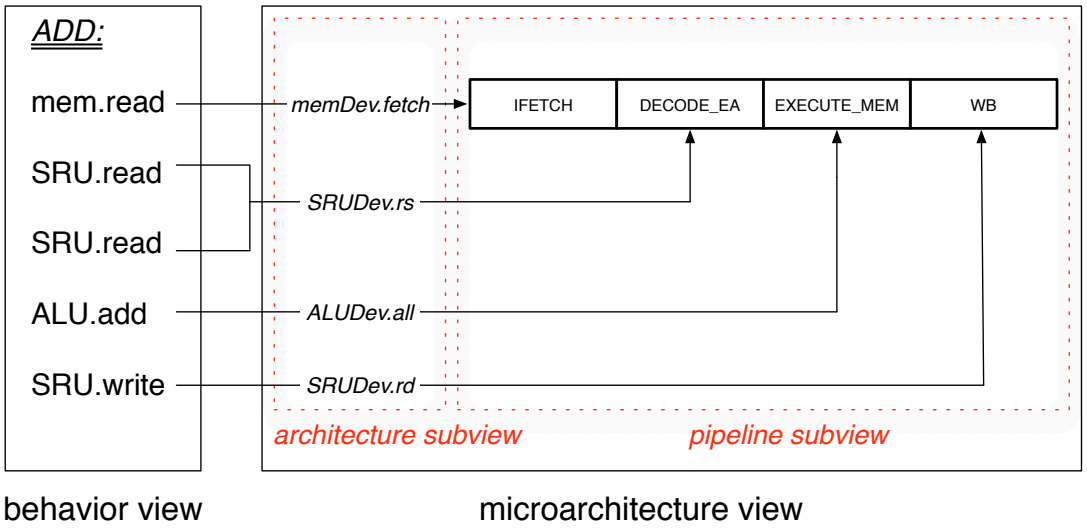


Figure 6: Mapping of the instruction set *behavior view* onto the *microarchitecture view* using component access. On the left, methods of components accessed by the `ADD` instruction are shown. They are mapped onto the 4-stage pipeline, using devices, and ports that control the concurrency between component accesses.

Sometimes, using any method of a component makes it unavailable. Instead of forcing the user to give the list of all methods, an empty list (without the ':') is interpreted as a full method list. The `ALUDev` device uses this scheme in line 13 in the example above .

The second object `pipeline` can be described as illustrated in the example below.

```
1 pipeline e200z1 maps to PPC5516 {
2
3   stage IFETCH {
4       memDev : fetch
5     }
6
```

```
7    stage DECODE_EA {
8        BPUDev : branch
9        SRUDev : rs
10       EAUDev : effAddrUnit
11     }
12
13   stage EXECUTE_MEM {
14       ALUDev : all
15       memDev : loadStore
16     }
17
18   stage WB {
19       SRUDev bypass in DECODE_EA : rd
20     }
21 }
```

In this example, the pipeline of the `e200z1` core is declared and mapped on the `PPC5516` architecture. In this object, the devices and their ports as well as the keywords (`read` and `write`) are used (see figure 6).

Contrary to Kassem et al. (2009a), there is no component method declared in the pipeline description, but devices' ports defined in the architecture subview. Moreover, the internal modeling of instruction has been enhanced. It allows both to take into account more complex instructions and to validate the correct mapping of instructions onto the pipeline, see section 7.1.1.

When using a port in a pipeline stage, it is implicitly taken at the start of the stage and released at the end of this stage. If a port needs to be held for more than one stage, the stage where it is released is explicitly given.

The data forwarding can be also expressed in Harmless. Data forwarding is

indicated by the keyword `bypass in` followed by a list of pipeline stages as shown at line 19 in the pipeline description above. Without the `bypass in`, forthcoming instruction would wait until the end of `WB` to start their `DECODE_EA`. With it, the result that will be written in port rd is available in the `DECODE_EA` stage of the next instruction. So, the next instruction in the pipeline, even if it is dependent, will not stall.

Let's consider a second microarchitecture based on the same ISA (PowerPC) to highlight the adaptability of Harmless: the e200z6 core (Freescale (2004)). Compared to the e200z1, devices and ports are unchanged but this core has a 7-stages pipeline with:

- 2 fetch stages;
- 1 decode stage, with the branch processing unit;
- 3 stages single path execute pipeline, with overlapped execution and feed-forwarding;
- 1 write-back stage.

As Harmless clearly splits the instruction set architecture from the microarchitecture view, only the pipeline sub-view should be adapted to get a model of the e200z6 core from the e200z1 core. The pipeline sub-view of this core is:

```
1 pipeline e200z6 maps to e200z6Core {
2   stage Fetch1 {
3       memDev : fetch
4     }
5
6   stage Fetch2 {
7     }
8
```

```
9   stage Decode { --For Branch Processing Unit
10      BPUDev : branch
11      SRUDev : rs
12      EAUDev : effAddrUnit
13    }
14
15  stage Execute1 {
16      ALUDev : all
17  }
18
19  stage Execute2 {
20      -- no port access
21  }
22
23  stage Execute3 {
24      memDev : loadStore
25  }
26
27  stage Register {
28      SRUDev bypass in Execute2 : rd
29    }
30 }
```

The 8-entry Branch Target Buffer is not yet taken into account.

As a result, splitting the ISA description in 3 views and using instructions signatures to bind a view to each others leads to a very flexible description. Having the microarchitecture described separately adds flexibility by allowing easily the retargeting of the ISA to a different microarchitecture.

5. Pipeline model

This section shows the internal model used for one of the most time-consuming hardware features to simulate: the pipeline. As Harmless does not mainly target any design space exploration, the internal pipeline model uses a finite state automaton and differs significantly from the real one. In this work, as a first step, we consider only sequential pipelines, what we call a simple pipeline (*i.e.* no pipelines working in parallel nor forking pipelines, that we consider as complex pipelines), but our final goal is to model any of them. Indeed, we consider that a complex pipeline is composed of a set of simple pipelines synchronized together and therefore a set of automata synchronized together. The choice of an automaton allows to move a part of the required computing time from runtime to build time (done only once, when generating the simulator). A state of the automaton represents the pipeline state at a given time (see figure 7): the instruction type is known for each stage of the pipeline. The link between the internal model and the high-level microarchitecture description (section 4.4) is explained in section 7.

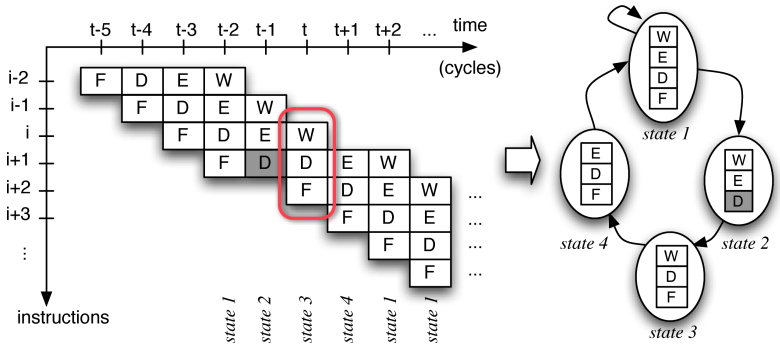


Figure 7: A state of the automaton represents the state of the pipeline at a given time. In this example with a 4-stage pipeline, three instructions are in the pipeline at time *t*, and the 'D' stage was stalled at time *t-1*. The automaton highlights the pipeline sequence, assuming that there is only one instruction type (restriction only for clarity reason).

At each clock cycle, the pipeline goes from one state to another depending on

a new instruction that enters the pipeline and on hazards. They are classified into three categories:

- *Structural hazards* arise from hardware resource conflicts when the hardware is needed by two or more instructions at the same time;
- *Data hazards* are the result of a data dependency between instructions;
- *Control hazards* occur when a branch is taken in the program.

Control hazards are resolved in the simulator at runtime: instructions that are in the delay slot of a taken branch instruction are dynamically replaced by stalls. In its current state, Harmless does not handle delayed branches and cancelled branches but this ability is currently considered. Constraints resulting from structural and data hazards are used to generate this automaton and modeled using *resources*.

5.1. Resources

Resources are defined as a mechanism to describe temporal constraints in the pipeline. They are used to take account of *structural hazards* and *data hazards*.

Two types of resources are defined, *internal* resources to model static constraints and *external* resources to model dynamic constraints.

5.1.1. Internal resources

They can be compared to *resources* in (Müller (1993)). They model structural hazards. An internal resource is fully managed by the pipeline, *i.e.*, the state of each *internal resource* (either *taken* or *available*) is fully defined by the pipeline state (each instruction is defined for each stage of the pipeline). In that case, when the automaton is built (and then the simulator), constraints described by *internal resources* are statically resolved when the set of next states is built. So, no computation overhead is required to check for this type of constraint at runtime.

For example, each pipeline stage is guarded by an *internal resource*. Each instruction that enters a stage takes the associated internal resource, and releases that resource when it leaves the stage. The resulting constraint is that each pipeline stage gets at most one instruction.

As a second example, consider a pipeline associated to an exclusive unified cache memory. Because concurrent accesses of instruction fetch and data read-/write are fully defined by the state of the pipeline, an internal resource is used.

5.1.2. External resources

They represent resources that are *shared* with other hardware components such as a system bus. It is an extension of internal resources to take into account resources that must be managed *dynamically* (*i.e.*, during the simulation). The state of each external resources is required to choose the appropriate transition to take.

For instance, in the case of a non exclusive memory controller, the pipeline is locked if it performs a request whereas the controller is busy. Otherwise, the pipeline stage that requests the memory access takes the resource. The memory controller is not exclusive if other hardware components can perform a memory access, such as a DMA or other peripherals.

An external resource is also used to check for data hazards. This external resource is called the *data dependency controller*. Let's consider a pipeline model with 4 stages (*Fetch*, *Decode*, *Execute* and *Write back*) and no data forwarding, as in figure 8. Operands are read in the *Decode* stage and written back in the *Write back* stage. When an instruction is in the *Fetch* stage (step ①), it checks that all required registers are available (registers that will be read in the next stage). This is one of the cases where the pipeline model differs from the real one. Indeed checking the availability of registers cannot be done before the Decode stage in the real pipeline. This operation updates the controller and its associated

external resource to either *available* or *busy*. The external resource state changes the next transition taken in the internal automaton and thus the pipeline model behavior: either it inserts a stall in the pipeline if the resource is busy, or it allows instruction in the *Fetch* stage to get in the next stage. When an instruction enters in the *Decode* stage, it first *locks* (step ②) registers that will be updated (registers with write access). These registers are *unlocked* in the *Write Back* stage (step ③).

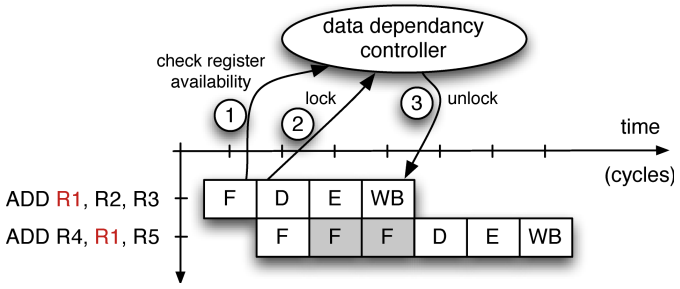


Figure 8: Interaction between the Data Dependency Controller and the pipeline model.

5.2. Instruction classes

To reduce the automaton state space, instructions that use the same resources (internal and external) are grouped into *instruction classes*.

The number of instruction classes is limited to $2^{R_{ext}+R_{int}}$ (R_{ext} and R_{int} are respectively the number of external and internal resources in the system), but this maximum is not reached because some *internal resources*, such as those preventing more than one instruction to enter a pipeline stage, are shared by all instructions, which leads to a lower number of instruction classes.

5.3. Internal finite automaton

The automaton represents all the possible simulation scenarios of the pipeline. A state of the automaton represents a state of the pipeline, which is defined as the list of all pairs (*instruction class, pipeline stage*) in the pipeline at a given time.

For a system with c *instruction classes*, there are $c + 1$ possible cases for each pipeline stage s of the pipeline (each instruction class or empty). The automaton is *finite* because it has at most $(c + 1)^s$ states. The initial state is the one that represents an empty pipeline (*i.e.* all stages are empty). A transition is taken at each clock cycle and its condition depends *only* on:

- the state of the external resources (busy or available);
- the *instruction class* of the next instruction that can be fetched in the pipeline.

Internal resources are already resolved in the generated automaton and do not appear in the conditions of the transitions. Other instructions in the pipeline are already known for a given automaton state, thus only the next instruction that will be fetched is necessary. This kind of transition condition is called a *basic* condition. As many different conditions can appear to go from one state to another, the transition condition is a disjunction of *basic* conditions.

The number of possible transitions is limited to at most $c \times 2^{R_{ext}}$ *for each state* (c is the number of *instruction classes* and R_{ext} is the number of external resources in the system). It implies that there are at most $(c+2)^s \times 2^{R_{ext}}$ transitions for the whole automaton.

6. Instruction Set Simulator generation

This section gives some details about the automatic simulator generation from the instruction set description. First, the instruction model is introduced. Then, the decoder generation is explained. Moreover, in order to speed up the simulation process, an improvement using a software-based instruction cache is presented. Finally, some results on the generation process are given for four processor descriptions.

6.1. Instruction Modeling

As indicated in section 4, in a Harmless description, an instruction corresponds to a path in the tree. This approach allows the sharing of common parts among different instructions. During the generation process, the tree structure will be flattened and the generator will duplicate the common parts of the description in the generated code for simulation efficiency (elimination of many function calls).

A *C++* class represents an instruction and offers three main methods: the *constructor*, the *execution* and the *mnemonic* functions.

6.1.1. The constructor

It is associated to the decode operation. Its goal is to identify the various fields of the instruction binary code (register index, immediate, address), and store values in the new object instance. The simulator context is never affected by this operation.

6.1.2. The mnemonic function

This function returns a string containing the instruction mnemonic. It is associated to the syntax description in Harmless (section 4.2) and used for disassembling. In the same way as the constructor, this function does not modify the simulator context. If no syntax description is given for an instruction, a default one is provided, returning the internal name of the instruction that is built from the signature.

6.1.3. The execution function

This function is in charge of simulating the instruction execution. It is directly linked to the instruction behavior description in Harmless (section 4.3). Using the previous example based on the addition instruction, this function will read the value of the 2 source registers, perform the addition and update the flag register,

and finally write back the result in the destination register. Even if the behavior description is split into multiple parts to take advantage of common behaviors, this function concatenates each part of the instruction behavior description, thus removing time-consuming function calls in the generated simulator.

Note that instructions that do not have any behavior description have a default execution that warns the user when executed. This is helpful when used in an incremental description approach because a simulator can be generated with an incomplete instruction set.

6.1.4. Classical interpretive execution approach

The execution of an instruction is initially based on an interpretive approach. The execution process is given in figure 9 and is done through five consecutive tasks. First, the decoder phase has to decode the binary code pointed by the program counter (explained in depth in section 6.2). Then, the instruction object is created (requiring a memory allocation) and fields are extracted from opcodes. After, the execution of the instruction is performed and finally the instruction object is deleted (requiring a memory deallocation). These steps have to be done for each instruction in the program, and memory allocation/deallocation are particularly penalizing in computation time. A more efficient approach based on a software-based instruction cache is explained in section 6.3.

6.2. Decoding phase

The decoder is generated using the format description part. It is an important part of the simulator generation. Since an instruction is represented as a branch in a tree, the first operation made is to flatten the tree and to extract all the format parts used in the description of each instruction. For each format, a couple mask/-value allows to identify the significant bits and the bits that are not representative. The condition (mask/value) is applied on the instruction binary code pointed by

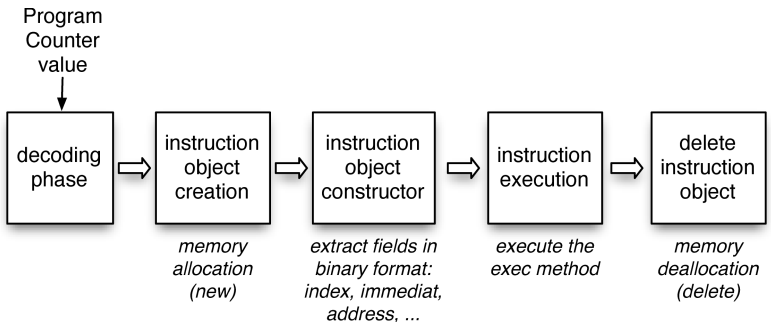


Figure 9: Different phases to execute one instruction, using an interpretative approach

the program counter.

In order to facilitate computations, the internal representation of conditions are encoded using *Binary Decision Diagrams* (BDD). This allows a very simple verification that the instruction codes do not conflict: if two instructions have the same binary code, then the conjunction of their BDD is not empty. Computing the conjunction of the BDDs for all combinations of two instructions is sufficient to verify that there is no code conflict in the instruction set.

With the internal use of BDDs, we obtain simple conditions, independently of the underlying description. This means that the description of the format tree has no influence on generated decoder performances, so many sub-nodes may be used for readability and easily reuse of common format parts, without simulation performance loss.

6.3. Efficient decoding phase using a software-based instruction cache

The execution process of the classical interpretive execution approach, presented in figure 9, has two major structural drawbacks:

- instructions of the embedded software under test inside a loop will be decoded many times;

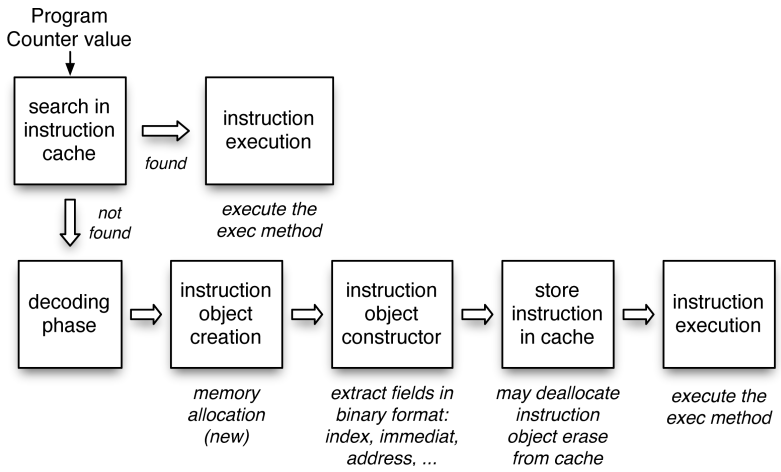


Figure 10: Execution of an instruction using the internal instruction cache.

- memory allocation/deallocation requires most of the computation time, when creating and deleting the C++ instruction object;

However, due to the presence of loops, the execution of a program has a strong temporal locality. To improve the previous approach, we added a software-based instruction cache during the decoding phase. This instruction cache have only an impact on computation performances, but does not change in any way the nature of the simulated hardware. This approach is a simpler version of the *Instruction Set Compiled Simulation* (Reshadi et al. (2003)). This cache is internal, it has only low side effects: the memory containing the program to simulate must not be modified during simulation because the change may not be taken into account, but we could consider this by disabling the corresponding instructions in the cache. This type of situation does not occur for our application field (embedded real-time systems), and this technique retains all the advantages related to the interpretive simulation approach (it does not depends on a specific program, as in the compiled simulation).

The simulation principle using a software-based instruction cache is described in figure 10. The first time the instruction is decoded, the software-based instruction cache returns a miss and a C++ object of the instruction is allocated as in the previous approach. The new instruction is stored in the cache (default cache size is 32K entries). The next time the instruction have to be executed, the instruction object is in the cache (the cache returns a hit). This way, the simulation time is reduced because, in the decoding phase, the object allocation and deallocation and the call to the constructor are removed.

In a Harmless simulator, the software-based instruction cache used is a direct-mapped cache. This is the easiest to implement and the speediest because there is no line lookup within the selected set. Experiences show the hit ratio is greater then 90%.

The internal software-based instruction cache is only intended to speed-up the computation time, this is *not* a model of a hardware cache of the CPU.

6.4. Results

In this section, we show some results about the simulator generation process for different processors:

- The *HCS12*, which is a *CISC* processor with a variable-size instruction set (from 1 to 8 bytes);
- The *PowerPC*, which is a *RISC* processor with a fixed instruction size (32 bits);
- The *ARM*, which is a *RISC* processor with a fixed instruction size (32 bits);
- The *Atmel AVR*, which is an 8-bit *RISC* processor (instruction length is 16 bits), even if some instructions use 32 bits.

These results give an overview of simulator performances and are available on table 1. A simple example, based on calculating a Fibonacci sequence, is simulated for each processor.

	HCS12	**PowerPC**	**ARM**	**AVR**
Description length (lines)	2925	3208	5122	1408
Instructions generated (nb)	5590	332	341	90
Generation time of the simulator source code from description (s)	30.4 s	4.3 s	3.5 s	0.4 s
Simulator source code size (C++ lines)	~ 418000	~ 41000	~ 128000	~ 12000
Time to compile the simulator (s)	545.2 s	32.4 s	116.9 s	11.9 s
Time to execute 100 millions of instructions of a basic example (s)	3.2 s	3.2 s	3.7 s	3.8 s
Time to execute 100 millions of instructions of a basic example without a software-based instruction cache (s)	18.6 s	14.6 s	20 s	14.2 s

Table 1: This table presents some results about the ISS simulator generation. Experiments done on an Intel Core 2 Duo @ 2 GHz (use of only one core)

The *RISC*-based architectures have fewer instructions than the *CISC* one that has more than 5500. The significant number of addressing modes is not the only reason, this is also due to the HCS12 architecture. Indeed, the HCS12 processor has only a few registers, and for instance, the instruction that rotates left is expanded into two instructions `ROLA` and `ROLB`, depending on the register considered (`A` or `B`). In the Harmless description, we have two possibilities to describe these instructions. The first one is to describe one instruction ROLx with one field parameter, and

the second one is to describe two different instructions. We chose the second one to take advantages of the internal software-based instruction cache: increasing the decoder complexity and simplifying the behavior description (and thus increasing the simulation speed).

As indicated above, the HCS12 model has more than 5500 instructions, this increases the time to generate the simulator and to compile the generated C++ files, but these processes are done only once. We can notice that if the verification of the conflicts in the instruction set is disabled (it is relevant only when describing the format part), the time to generate simulator sources is reduced to 18.7 s. Most ARM instructions are conditional, which leads to increase the code of the behavior of each instruction. An ISS simulator is built in less than 10 minutes for a *CISC*-based architecture and in less than 2 minutes in the case of a *RISC*-based architecture.

The simulator generation process is executed once for each model. So the most important point to compare is simulation speed, which refers to the last two lines of the table 1. We can notice that the computation time required for the four models are in the same order of magnitude (less than 4 s with a software-based instruction cache and 20 s without cache). This result shows the importance of a software-based instruction cache that can prevent multiple decodings of the same instruction: the suppression of the software-based instruction cache leads to a loss of 80% in performance.

Finally, the results show the good performance of simulators generated automatically from the description of multiple processors (*HCS12*, *PowerPC*, *ARM* and *AVR*). Other simulations are presented in section 7.3, based on the Mibench benchmark suite.

7. Cycle Accurate Simulator generation

This section presents the different steps of the CAS simulator generation from the description of both the ISA and the processor microarchitecture.

This section first bridges the gap between the microarchitecture description (section 4.4) and the internal pipeline model (section 5). Then it focuses on the generation of the internal representation of a sequential pipeline which is a finite-state automaton. This finite-state automaton generated by *p2a* is used as an input of the *a2cpp* tool. *a2cpp* generates the *C++* code simulating the pipeline mechanism. Eventually, some results on the simulator generation process from several descriptions of processors are presented and analyzed.

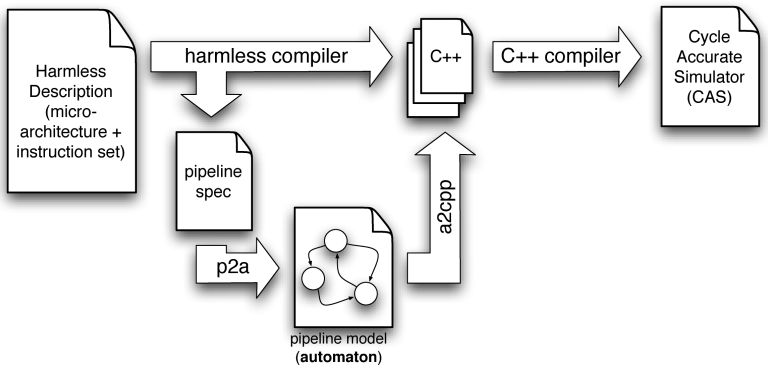


Figure 11: *CAS* simulator development chain. Here, we focus on the compiler part that deals with the generation of the pipeline specification file and the *p2a* tool that generates the finite-state automaton from a pipeline specification.

7.1. Generation of the intermediate pipeline representation

The internal pipeline model presented in section 5 is based on several characteristics such as *pipeline stages*, *instruction classes* and constraints through *internal* and *external resources*, while the microarchitecture is described with the *architecture* view.

First, the generation of *instruction classes* is explained. Then we present the instruction execution principle in the different stages of the pipeline.

7.1.1. Instruction class generation

Starting from the *architecture* and *behavior* views, instructions are grouped into classes to reduce the automaton size. As indicated above, an instruction class gathers instructions that use the same *port* (shared or not) the same number of times at the same pipeline stage. Each *port* corresponds to an *internal resource* and each *shared port* corresponds to an *external resource*.

The instruction class generation is done when mapping instructions onto the pipeline as in figure 6. The Harmless compiler first builds an oriented graph of the components' methods used for each instruction, using the behavior view of the ISA description.

Then, it matches each graph to the oriented graph of the devices' ports defined in the *architecture* subview, using the pipeline description (algorithm 1). The

algorithm recursively explore the oriented graph.

Algorithm 1: The simplified algorithm to map one instruction onto the pipeline *mapInstructionOnPipeline*. The device/port oriented graph is built when mapping the instruction onto the pipeline. Initial parameters are the list of pipeline stages and the root node of the component's method oriented graph.

```
input  : stageList // a list of pipeline stages
input  : instructionNode // a node of the oriented graph of components'
         methods used by instruction
output : bool: mappingFound
bool foundStage ← false
for each pipeline stage in stageList while not foundStage do
    foundStage ← component's method of instructionNode matches one port in
    current pipeline stage and this port can be taken
    if not foundStage then
        remove the stage in stageList;
if foundStage then // start recursion and set device node
    bool subOk ← true
    // explore subnodes, starting from the current stage (recursion)
    for subnodes of instructionNode while subOk do
        mappingFound ← mapInstructionOnPipeline(stageList, subnode)
    add the node to the graph of devices' ports accesses.
else
    Report an error: "The instruction cannot map onto the pipeline"
```

It allows to verify that components' methods are accessed in a correct order so that the ISA can be mapped onto the pipeline.

To flatten the oriented graph of ports, a port is added in the list of the required port access if it is present in at least one branch of the graph.

Here is a partial description of the PowerPC `addi` instruction (`add` with `i`mmediat) to highlights the mapping. The contents of register `R0` is hardwired to 0:

```
1  if rA != 0 then
```

```
2       op1 := SRU.GPR.read32(rA)
3     else
4       op1 := 0
5     end if
6     -- add with imm
7     result := ALU.addInt(op1 , SIMM)
8     SRU.GPR.write32(rD, result)
```

A graphical representation of the oriented graph extracted from the behavior of the `ADDI` is in the figure 12, including the instruction fetch in memory.

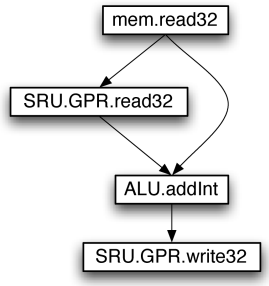


Figure 12: Oriented graph to model the instruction access to components' methods of the instruction `ADDI`

The corresponding oriented graph of ports usage is in figure 13. It shows all the valid paths available for the pipeline (model `e200z1` defined in section 4.4), and highlights in bold the graph of the `ADDI` instruction.

Then, for the `ADDI` instruction, the following ports are used: `memDev.fetch`, `SRUDev.rs`, `ALUDev.all` and `SRUDev.rd`. The port `SRUDev.rs` is present in only one branch but it is put in the list of the required port when paths of the graph are merged. Here the special case of register `R0` is removed. The Instructions that are using the same ports at the same pipeline stage and the same number of times than this instruction are grouped into the same *instruction class*.

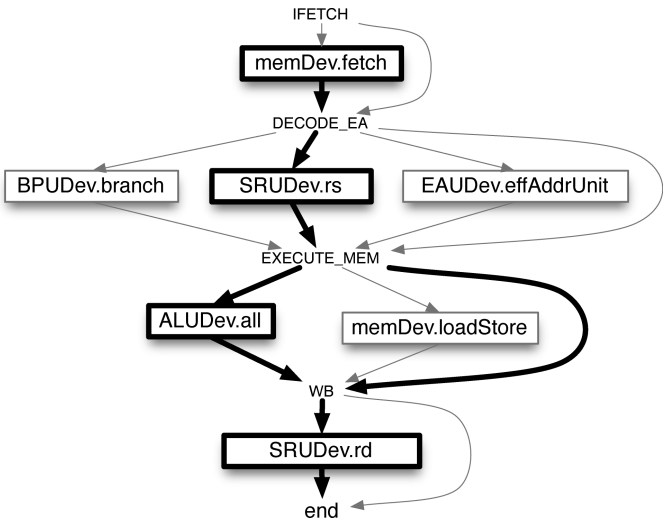


Figure 13: Oriented graph to model the valid paths (devices/ports) of instructions. The path of the instruction `ADDI` is highlighted in bold.

The mapping of the instructions onto the pipeline described in this section differs from the one presented in Kassem et al. (2009a). The latter does not use any oriented graph, but only a flat list of component's methods accesses. Consequently, it does not allow to check that the instruction description maps correctly onto the pipeline.

In the description of the *PowerPC 5516* microarchitecture (see section 4.4) there are 2 shared ports `loadStore` and `fetch` (which are translated to external resources). In addition, 1 external resource is used to check for data dependencies during simulation. Other ports are interpreted as internal resources. In the end, 12 resources are used:

- 4 for the pipeline stages (One for each stage);
- 1 for the integer unit management: `all`;
- 2 to constrain the accesses to System Register Unit's (SRU) methods: `rs` and `rd`;

- 1 for the Branch Processing Unit (BPU): `branch`;
- 1 for the Effective Address Unit (EAU): `effAddrUnit`;
- 2 for the memory accesses (external): `fetch` and `loadStore`;
- 1 resource to check data dependencies (implicit and external): `dataDep`.

For this model, 21 instruction classes are generated.

Reduction of the number of instruction classes. The size of the automaton (the execution model of the pipeline) increases polynomially with the number of instruction classes (see section 5.3). This often causes a combinatorial explosion. Therefore it is interesting to minimize this number.

The number of external resources cannot be reduced because they can be taken by other hardware components that operate concurrently, such as a *memory controller*. These resources rule the dynamic constraints that will be solved during the simulation. So the reduction of instruction classes focuses only on internal resources by eliminating private ports (not *shared*) that do not represent constraints on the use of component methods.

For example, let's consider a processor in which the register bank allow n accesses in parallel. If we consider all the combinations of instructions in the pipeline, we can get the combination that requires the maximum of parallel accesses m that may be done. If this maximum is equal or lower than authorized accesses in description, there is no constraint and the associated internal resource can be removed at compile time.

We distinguish two cases for performance reasons:

- if the internal resource can be taken in only one stage of the pipeline, the resource can be deleted if its usage by all instructions (which require this resource) does not exceed the usage permitted in the *architecture* subview;

- a general case where it is necessary to test all possible combinations of instructions in execution state in the pipeline (their number is given by the mathematical arrangement A_c^s, where s is the number of pipeline stages and c the number of instruction classes), to see if, at any time, the port usage specified in the *architecture* subview does not exceed the port capacity (*i.e.* no structural hazard can occur). The resource is removed in that last case.

Let's consider again the example that describes the microarchitecture of the *PPC5516* processor composed of a 4-stage pipeline `e200z1` mapped onto the architecture named `PPC5516`. Using this model, the 21 instruction classes are reduced to 2 instruction classes. Indeed, before the reduction, 5 ports (not shared) are needed (`all`, `rs`, `rd`, `effAddrUnit` and `branch`). After the reduction, none of the unshared ports remains

7.1.2. Internal automaton generation

The internal automaton represents all the possible simulation scenarios of the pipeline (see section 5.3). The generation of this automaton is performed by the external tool *p2a*. *p2a* takes as input the pipeline specifications based on pipeline stages, the instruction classes and the resources that have been extracted and minimized from the Harmless high level description. Here is presented the generator algorithm 2, based on a breadth-first exploration graph.

At start the states list contains the initial state of the automaton which is an empty pipeline. From that initial state, the algorithm computes all the possible basic conditions and get the set of next states. New next states are added to the state list. The algorithm does the same computation for each state of the states list. It stops when there is no new state to add to the states list.

The main function of this algorithm is the one that can *get the next automaton state*, when a basic condition is known. From a generic pipeline model, this

Algorithm 2: Generation of the automaton pipeline model.

- Create a list that contains the initial automaton state;
- Create an automaton, with the initial automaton state;

while *list is not empty* **do**

- Get an automaton state in the list (start state);
- Generate all the possible basic conditions (combinations of external resources, combined with the instruction class of the next instruction fetched);

for *each basic condition* **do**

- *Get the next automaton state* (this is a deterministic automaton), using the basic condition and the start state;

if *the state is not yet included in the automaton* **then**

- Add the new automaton state (target state) in the list;
- Add the new automaton state in the automaton;

- Update (or create) the transition's condition, by adding a basic condition (disjunct);

- Remove automaton start state from the list;

function computes the next state of the automaton, taking account all the constraints brought by resources (internal and external) as shown in the algorithm 3. A pipeline is modeled as an ordered list of pipeline stages, where each pipeline stage is an internal resource. In this algorithm, the pipeline stages in the loop are taken from the last to the first, because the pipeline stage that follows the current one must be empty to get a new instruction. This algorithm allows to detect sink states in the automaton (not shown in the algorithm 3, for clarity reason). A sink state corresponds to deadlock in pipeline and consequently to a wrong description.

Algorithm 3: Function that *gets the next automaton state*, from a given state, with a known basic condition.

```
for each pipeline stage, from the last to the first do
    if there is an instruction class in the current stage then
        if resources required by the instruction class can be taken in the next
        pipeline stage then
            - Instruction class releases resources in the current pipeline stage;
            if there is a next pipeline stage then
                - Instruction class is moved in the next pipeline stage;
                - Resources required in the next pipeline stage are taken;
            - Instruction class is removed from the current stage;
```

Combinatorial explosion There is a combinatorial explosion when the complexity of the modeled pipeline increases. As presented in section 5.3, the automaton is bounded by $(c+1)^s$ states and $(c+2)^s \times 2^{R_{ext}}$ transitions. The maximum size of the automaton increases exponentially with the pipeline depth and the number of external resources, and in a polynomial way with the number of instruction classes. For short pipelines (typically 5 to 6 stages), no combinatorial explosion is observed. For deeper pipelines, the automaton may be too big. One of the solution

to solve this problem is to split the pipeline into two or more parts to generate two or more smaller automata that are synchronized using external resources.

7.2. Results

This section shows some results about the generation process and execution of both *ISS* and *CAS* simulators, as well as a comparison with a real target processor. Eventually, the simulators generated by Harmless are compared to other existing simulators.

All simulations are done on an Intel Core2Duo@2.4 Ghz processor. The simulator is single threaded and uses only one core.

7.2.1. Generation process

We focus in this section on two widely used instruction sets: ARM(v5) and PowerPC. The generation of the simulator sources from the Harmless description is not significant (respectively 9.6s and 4.6s) and the compilation of the simulator binary takes 140.6s and 59.5s. As these generated simulators are interpreted simulators, they do not depend on the simulated application software. The generated process is run only once.

We consider the 2 cycle accurate models defined in section 4.4. The first one is the *PowerPC 5516* from *Freescale*, with a e200z1 core (Freescale (2008)) having a 4-stages pipeline and the second is a e200z6 core (Freescale (2004)) with a 7-stages pipeline.

The generation of the simulator sources from the description of the e200z1 and e200z6 models take 7.8s and 12.9s respectively. The automata modeling these pipelines have 1405 and 26 913 states respectively. The simulator binary is built from its C++ sources in 79s and 81s. This is quite acceptable as simulators are built only once.

7.3. Benchmark

We ran the *Automotive and Industrial Control* MiBench benchmark suite (Guthaus et al. (2001)). Even if these benchmarks are dedicated to the automotive context, they require the use of a filesystem; Harmless has been updated to allow the use of the host filesystem through stubs. Figure 14 shows the results on the two ISS and the two CAS models.

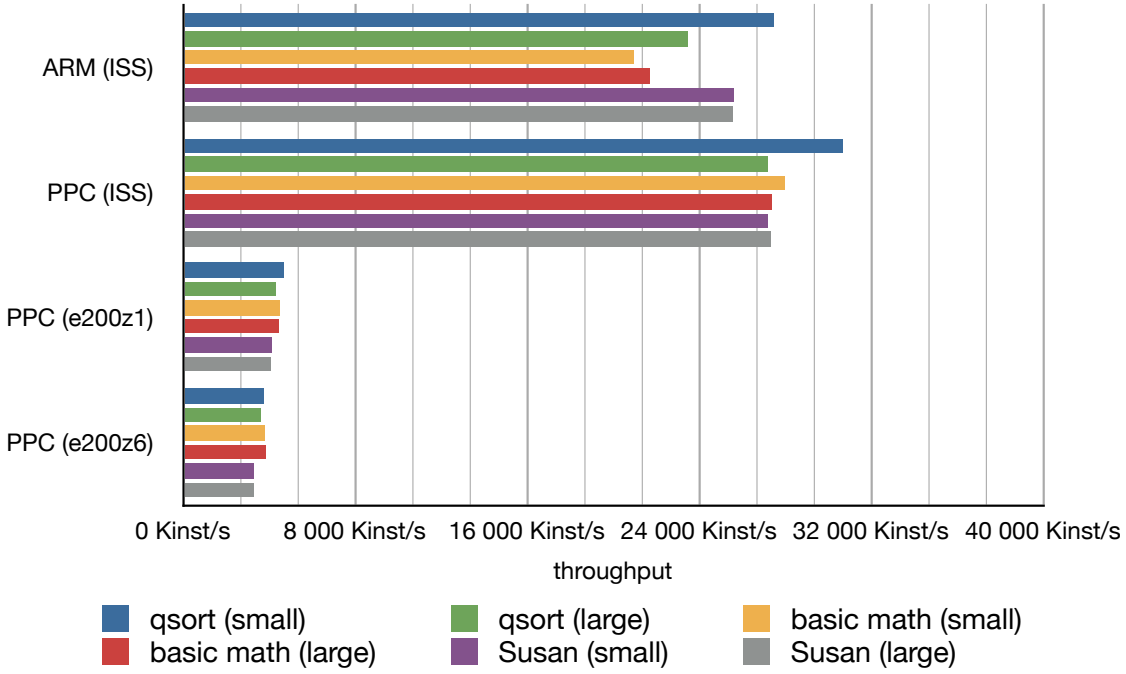


Figure 14: Mibench Automotive benchmark. It shows the instruction throughput in Kinst/s.

It shows that the ISS models are really faster than CAS ones, with an average throughput of 24 000 Kinst/s for the ARM model and 28 000 Kinst/s for the PowerPC. It compares favorably to other existing simulators. In (Ratsiambahotra et al. (2009)), T. Ratsiambahotra and al. compared their automatically generated simulator (using the gliss ADL, based on nML) to other existing ones, using the same benchmark. Measures were done on a DualCore@3GHz. They get an average throughput of 6500 Kinst/s for simplesim, the ARM functional simulator of sim-

plescalar 4.0 (which is largely used in the architecture research community), and 7800 Kinst/s for the gdb-armulator 6.7, another hand-written functional simulator. With Gliss, the average throughput was 8500 Kinst/s.

We compared our simulator with a PowerPC ISS simulator generated using ArchC, using only the "BasicMath" MiBench benchmark because others require a filesystem access. The PowerPC ISS generated using ArchC runs at 38 800 Kinst/s (37 900 Kinst/s for BasicMath small) while our Harmless PowerPC ISS runs at 27 900 Kinst/s for the same benchmarks, which makes ArchC 39% faster. However, the ArchC description is limited to the decoding phase of instruction and the rest of the simulator is hand coded, including instruction behavior and simulation engine.

Two ARM simulators are provided with the MADL package. Like ArchC, we ran only the "BasicMath" MiBench benchmark for same reasons. The ARM ISS model generated by MADL run at 23 400 Kinst/s while the Harmless ISS model is at 20 000 Kinst/s. MADL is 17% faster, but some parts of the ISS are also hand-written (simulation engine). This makes this ISS in an intermediate stage between an hand-written ISS and a fully generated one. The CAS is a StrongARM-1100 (in-order five-stage classic RISC pipeline) that run the same benchmark program at 3900 KCycles/s. We modeled the same pipeline architecture on Harmless and got a throughput of 4200 KCycles/s. Harmless is 8% faster but does not model the memory hierarchy. Simulation times are in the same order of magnitude.

The Harmless CAS models are nearly 6.5 (e200z1) and 7.8 (e200z6) times slower than the ISS PowerPC model. Figure 15 compares the throughput of the two CAS in KCycles/s. Even if the e200z1 has an instruction throughput slightly higher than the e200z6, the latter has an higher throughput in cycles. For a given number of instructions, the model with the shortest pipeline (e200z1) is the fastest, but it also requires less cycles for the same number of instructions. The computation

time required are in the same order of magnitude (about 5600 KCycles/s).

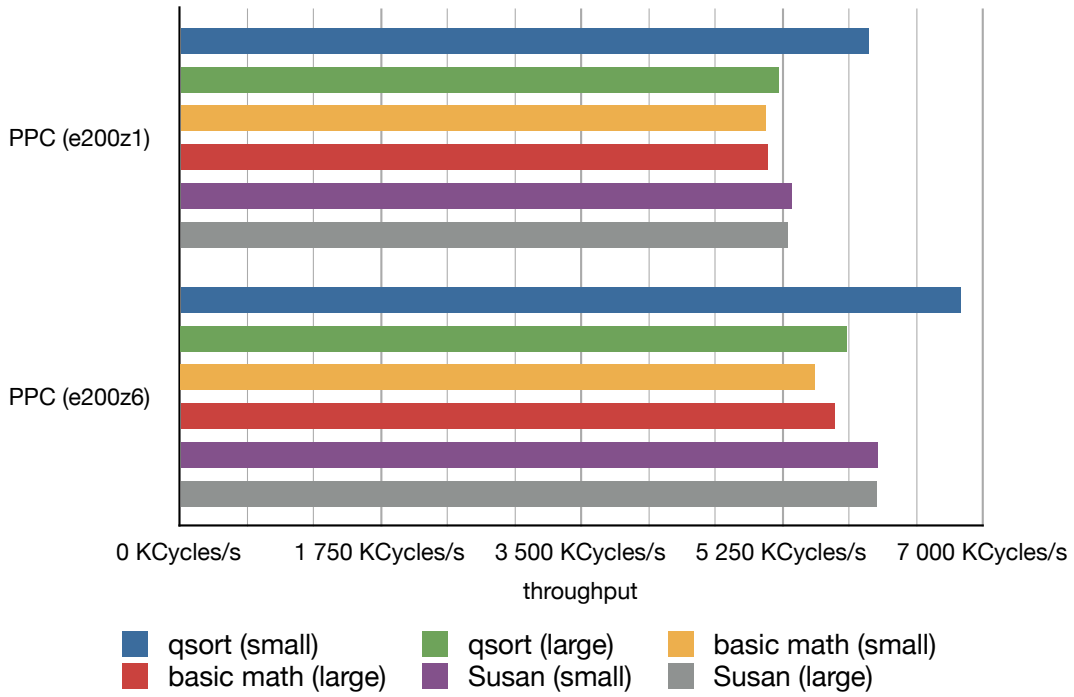


Figure 15: Mibench Automotive benchmark. Throughput of CAS models in KCycles/s.

We can notice that the pipeline depth (from 4 stages to 7 stages) have a very low influence on the simulation computation time. This is largely due to the internal model of the pipeline that resolves an important part of dependencies (only 1 instruction per stage for instance) directly during the automaton generation.

7.3.1. Comparison with the real target

This section compares the *e200z1* model to the real target, the *PowerPC 5516*. The clock frequency of the processor used for the test is 16 MHz. The *PowerPC 5516* is targeted to the automotive market and features a e200z1 processing core. The core is connected to an internal SRAM and to an internal Flash memory by using a crossbar switch. SRAM and Flash memories may be accessed in parallel

without timing penalty. The model has been written according to the e200z1 Reference Manual (Freescale (2008)).

The Mibench are not suitable in this context, as the real target does not have any filesystem. We used a simple example based on *Trampoline*, a Real-Time Operating System (RTOS)[3]. The example is composed of 3 tasks that activate each others in a cyclic pattern. It produces numerous preemptions. Most of the code is composed of system calls of the *RTOS*. The program is put in the flash memory and the data in the SRAM memory.

	Application execution time (in cycles)	**CPI**
Model	*42014*	*1.57*
Real target	*44110*	*1.65*
Error percentage	4.75%	

Table 2: Comparison of the e200z1 PowerPC core model with the real target, in cycles. CPI stands for Cycles Per Instruction.

The time taken for a cycle was measured on the real target and the same has been computed using the Harmless simulator. Table 2 gives an overview of the simulator accuracy compared to the real target. We can notice that the *CAS* simulator accuracy, generated from the *PowerPC 5516* processor description in Harmless, is good and very close to reality: the obtained error is less than 5% for both measurements, which is widely acceptable. The difference comes from undocumented timings of the e200z1 core that are not modeled in Harmless. For instance, two loads from memory using the same register as target incur a 1 cycle stall in the pipeline, a case that is not documented. However, the model could

[3] *Trampoline* is a (*RTOS*) distributed as free software, compatible with the automotive standard *AUTOSAR 3.1 SC 4* (`http://trampoline.rts-software.org/`).

be tuned to encompass the undocumented timings. By using special purpose programs to measure the execution time of sequences of instructions, the documented timings can be checked and the undocumented ones can be brought out.

8. Conclusion

As the complexity and sophistication of real-time embedded systems increase, validation tools at each step of the design cycle are more and more needed. This is also true for the simulation of the actual code of embedded applications. Development of hardware simulators by hand is a lengthy, difficult and error-prone task, especially for complex pipelined processors.

Providing a quick and efficient means to generate a simulator of the hardware platform requires to use an ADL. The design of Harmless addresses this issue. Harmless has been designed to describe a processor in a modular way. The description of the instruction set uses 3 decoupled views (format, behavior and syntax) to allow to choose the best description for each view. This part of the description is used to generate an ISS. A fourth view describes the microarchitecture: pipeline and component access constraints. Having a separate view is more flexible as it enables to describe several microarchitectures for the same instruction set.

The microarchitecture view and the behavior view are used to synthesize a finite-state automaton where a state is the state of the pipeline at a given time. Experiments on the CAS show that this pipeline model limits the impact of the pipeline depth on the computation time.

Currently, a Harmless compiler exists and generates an instruction set simulator or a cycle accurate simulator. The performances of the simulators are good and compare favorably to existing simulators. A prototype of Harmless compiler can be downloaded at `http://harmless.rts-software.org`.

Future work will focus on the minimization of the automaton, the use of mul-

tiple automata to model and simulate superscalar processors. How to model dynamic superscalar processors, including speculative execution is also planned. At last, device modeling will be investigated to generate simulators of a complete microcontroller.

References

Azevedo, R., Rigo, S., Bartholomeu, M., Araujo, G., de Araujo, C. C., Barros, E., 2005. The archc architecture description language and tools. International Journal of Parallel Programming 33 (5), 453–484.

Bashford, S., Bieker, U., Harking, B., Leupers, R., Marwedel, P., Neumann, A., Voggenauer, D., 1994. The mimola language version 4.1. Tech. rep., Lehrstuhl Informatik XII University of Dortmund, Dortmund.

Béchennec, J.-L., Briday, M., Alibert, V., june 2011. Extending harmless architecture description language for embedded real-time systems validation. In: 6th IEEE International Symposium on Industrial Embedded Systems (SIES'11), Västerås, Sweden.

Fauth, A., Knoll, A., 1993. Automated generation of dsp program development tools using a machine description formalism. In: International Conference on Acoustics, Speech and Signal Processing (ICASSP). IEEE, pp. 457–460.

Fauth, A., Praet, J. V., Freericks, M., 1995. Describing instruction set processors using nml. In: EDTC '95: Proceedings of the 1995 European conference on Design and Test. IEEE Computer Society, Washington, DC, USA, p. 503.

Freericks, M., 1991. The nml machine description formalism. Tech. Rep. 1991/15, Computer science department, TU Berlin, Germany.

Freescale, 2004. e200z6 PowerPC Core Reference Manual. Freescale Semiconductor, Inc.

Freescale (Ed.), 2005. Programming Environments Manual for 32-Bit Implementations of the PowerPCTM Architecture. Freescale semiconductor.

Freescale, 2008. e200z1 Power Architecture Core Reference Manual. Freescale Semiconductor, Inc.

Guthaus, M. R., Ringenberg, J. S., Ernst, D., Austin, T. M., Mudge, T., Brown, R. B., 2001. Mibench: A free, commercially representative embedded benchmark suite. In: Proceedings of the Workload Characterization, 2001. WWC-4. 2001 IEEE International Workshop. IEEE Computer Society, Washington, DC, USA, pp. 3–14.

Hadjiyiannis, G., Hanono, S., Devadas, S., 1997. Isdl: an instruction set description language for retargetability. In: DAC '97: Proceedings of the 34th annual conference on Design automation. ACM, New York, NY, USA, pp. 299–302.

Hadjiyiannis, G., Hanono, S., Devadas, S., september 2000. Isdl: An instruction set description language for retargetability and architecture exploration. Design Automation for Embedded Systems, Springer 6 (1), 39–69.

Hadjiyiannis, G., Russo, P., Devadas, S., 1999. A methodology for accurate performance evaluation in architecture exploration. In: DAC '99: Proceedings of the 36th annual ACM/IEEE Design Automation Conference. ACM, New York, NY, USA, pp. 927–932.

Halambi, A., Grun, P., al., March 1999. Expression: A language for architecture exploration through compiler/simulator retargetability. In: European Conference on Design, Automation and Test (DATE), Munich, Germany. pp. 485–490.

Hanono, S., Devadas, S., 1998. Instruction selection, resource allocation, and scheduling in the aviv retargetable code generator. In: DAC '98: Proceedings of the 35th annual Design Automation Conference. ACM, New York, NY, USA, pp. 510–515.

Hoffmann, A., Meyr, H., Leupers, R., 2002. Architecture Exploration for Embedded Processors with LISA. Kluwer Acaddemic Publishers.

Kassem, R., Briday, M., Béchennec, J.-L., Savaton, G., Trinquet, Y., September 2009a. Cycle accurate simulator generation using harmless. In: Eurosis (Ed.), International Middle Eastern Multiconference on Simulation and Modelling (MESM'09), Beirut, Lebanon.

Kassem, R., Briday, M., Béchennec, J.-L., Trinquet, Y., Savaton, G., March 2009b. Instruction set simulator generation using harmless, a new hardware architecture description language. In: 2nd International Conference on Simulation Tools and Techniques Simutools'09.

Kranen, K., july 2006. `SystemC` 2.2.1 User's Guide. Synopsys, Inc.

Lohr, F., Fauth, A., Freericks, M., 1993. Sigh/sim – an environment fo retargetable instruction set simulation. Tech. Rep. 1993/43, Computer science department, TU Berlin, Germany.

Mishra, P., Dutt, N. (Eds.), 2008. Processor description languages. Morgan Kaufmann Publishers.

Müller, T., 1993. Employing finite automata for resource scheduling. In: MICRO 26: Proceedings of the 26th annual international symposium on Microarchitecture. IEEE Computer Society Press, Los Alamitos, CA, USA, pp. 12–20.

Paakki, J., 1995. Attribute grammar paradigms—a high-level methodology in language implementation. ACM Comput. Surv. 27 (2), 196–255.

Panda, P. R., 2001. SystemC: A modeling platform supporting multiple design abstractions. International Symposium on System Synthesis 0, 75–80.

Pees, S., Hoffmann, A., Zivojnovic, V., Meyr, H., 1999. Lisa - machine description language for cycle-accurate models of programmable dsp architectures. In: DAC '99: Proceedings of the 36th ACM/IEEE conference on Design automation. ACM, New York, NY, USA, pp. 933–938.

Qin, W., 2004. Modeling and description of embedded processors for the development of software tools. Ph.D. thesis, Princeton University, Princeton, NJ.

Qin, W., Rajagopalan, S., Malik, S., June 2004. A formal concurrency model based architecture description language for synthesis of software development tools. In: Conference on Languages, Compilers, and Tools for Embedded Systems (LCTES'04). Washington, DC, USA.

Rajesh, V., Moona, R., 1999. Processor modeling for hardware software codesign. In: VLSID '99: Proceedings of the 12th International Conference on VLSI Design - 'VLSI for the Information Appliance'. IEEE Computer Society, Washington, DC, USA, p. 132.

Ratsiambahotra, T., Cassé, H., Sainrat, P., 2009. A versatile generator of instruction set simulators and disassemblers. In: SPECTS'09: Proceedings of the 12th international conference on Symposium on Performance Evaluation of Computer & Telecommunication Systems. IEEE Press, Piscataway, NJ, USA, pp. 65–72.

Reshadi, M., Mishra, P., Dutt, N., 2003. Instruction set compiled simulation: a technique for fast and flexible instruction set simulation. In: DAC '03: Proceed-

ings of the 40th annual Design Automation Conference. ACM, New York, NY, USA, pp. 758–763.

Target, June 2003. Chess/checkers: a retargetable tool-suite for embedded processors. Technical white paper, retreived on *http://www.retarget.com*.

Zimmermann, G., 1979. The mimola design system a computer aided digital processor design method. In: DAC '79: Proceedings of the 16th Design Automation Conference. IEEE Press, Piscataway, NJ, USA, pp. 53–58.

Zivojnovic, V., Pees, S., Meyr, H., 1996. LISA - machine description language and generic machine model for hw/sw co-design. In: IEEE Workshop on VLSI Signal Processing. pp. 127–136.