CYCLE ACCURATE SIMULATOR GENERATION USING HARMLESS*

Rola Kassem, Mikaël Briday,
Jean-Luc Béchennec and Yvon Trinquet
IRCCyN
1, rue de la Noé BP 92101
44321 Nantes Cedex 3, France
email: firstname.name@irccyn.ec-nantes.fr

Guillaume Savaton
ESEO
4, rue Merlet de la Boulaye - BP30926
49009 Angers Cedex 01, France
email: guillaume.savaton@eseo.fr

KEYWORDS
Cycle Accurate Simulator, Hardware Architecture Description Language, Pipeline Modeling.

ABSTRACT
Simulators are commonly used in embedded system development processes for early functional validation of code and exploration of new instruction set design. Such a simulator can be either hand-written or generated automatically, based on an Hardware Architecture Description Language. Automatically generated simulators are more maintainable and are faster to develop, but they also generally suffer from low performances in simulation speed and a lack of expressivity in the description. This paper presents how a cycle accurate simulator is generated automatically from a description written using the HARMLESS language. It differs from other hardware description languages in many ways: it resolves most expressivity issues and naturally offers a flexible description by explicitly splitting the syntax (mnemonic), format (binary code) and behavior descriptions. Thus, it allows an incremental description, starting for example by the disassembler (requiring format and syntax descriptions). When the first two descriptions are validated, the behavior description is added to obtain the instruction set simulator. Finally, the micro-architecture description adds information to build a cycle accurate simulator.

Introduction
The interest of the simulation techniques for the design of software, in particular for embedded systems, does not need to be justified anymore. Simulation techniques do not oppose to other V&V techniques (Verification & Validation), particularly formal V&V techniques; they are complementary of these techniques often based on large grain models. Another interest of simulation is the possibility to design and validate software when the hardware is not yet available. In this way, both software and hardware may be designed simultaneously and time-to-market is reduced.

The simulator is the central element and its specifications depend on the simulation objectives. In our application field - real-time embedded systems - we need to simulate the execution of the binary code of an application on a fine grain model of the processor. Two simulation approaches are attractive. The Instruction Set Simulator (ISS) only takes into account the instruction behavior, independently of the time needed to execute the instruction, whereas a Cycle-Accurate Simulator (CAS) takes into account 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 while ISS has better performances. Some ISS can also be associated to a structural simulator to offer both ISS and CAS advantages. The development of an ad hoc simulator is a difficult task, especially the simulator validation for complex modern processors. Moreover, most of this work is not reusable for a new target architecture. So, in order to simplify this task, an hardware Architecture Description Language (ADL) may be used.

The work presented in this paper is a part of a larger project aiming to the realization of a simulator offering both ISS and CAS possibilities, and used for the design of real-time embedded systems. Some information about ISS and CAS simulator generation are quickly given in this paper and the interested reader can refer to [Kassem et al. (2008)] and [Kassem et al. (2009)] for more details. This paper focuses on the automatic generation of a Cycle-Accurate Simulator from the description of the pipeline using HARMLESS (Hardware ARchitecture Modeling Language for Embedded Software Simulation), an Architecture Description Language (ADL). HARMLESS differentiates itself from other ADL by using four independent views for binary format, syntax, behavior and micro-architecture. This approach is more flexible and eases the description. Some internal aspects on the decoder generation and the execution approach are also shown.

The paper is organized as follows: The first section is dedicated to related work on other ADL with an emphasis on LISA. The second section explains how the pipeline is modeled internally to generate the simula-

*This work is supported by ANR (French National Research Agency) under Grant Number ADEME-05-03-C0121.
tor. The third section gives an overview of the language and focuses on the micro-architecture view. In the next section, some internal aspects of HARMLESS ADL are given. Then some results are presented and the conclusion ends the paper.

**Related work**

Generally, an ADL must be able to specify a wide variety of architectures. An architecture description must be easy to modify and easy to understand by a designer. A lot of works has been done on ADL that can be classified into three categories: the first one that captures the Instruction Set (i.e., nML [Fauth et al. (1995)], ISDL [Hadjiyiannis et al. (1997)]); the second one that captures the structure of the processor (i.e., MIMOLA [Bashford et al. (1994)]); and the third one that mixes the two first categories (IS and structure) (i.e., LISA [Pees et al. (1999)], EXPRESSION [Halambi et al. (1999)]). To obtain an ISS, an ADL must allow the description of the binary format, the behavior and the syntax of the instructions. In addition, in case of a CAS, it is necessary to add the description of the micro-architecture.

ISDL [Hadjiyiannis et al. (1997)] is more flexible and its semantic is stronger than those of nML. It allows the description of a wide variety of architectures with emphasis on VLIW architectures. As nML, The IS contains behavioral and structural informations. Moreover, it allows the description of the hardware of micro-architecture, that is mixed with the behavior of the instructions.

The main goal of MIMOLA [Bashford et al. (1994)] is the hardware-software co-design, this implies a much complete modeling of the structure of the processor, and gives the advantage that the same description is used for both processor synthesis, and code generation. The IS is extracted from the structure, this task can be difficult for complex instructions. Generally, MIMOLA is considered as a very low-level language and is laborious to write and modify. In addition, simulation is slow in MIMOLA environment.

In [Pees et al. (1999)], LISA is presented as a machine description language that gives a formal description of programmable architectures, their interfaces and peripherals. In many aspects, LISA includes ideas which are analogue to 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) and, from this point of view, is close to HARMLESS. However, HARMLESS uses a different approach. In both languages, the designer defines the pipeline stages. With LISA, pipeline registers are explicitly defined and instructions reference these registers in their behavior. As a result a change in the pipeline implementation leads to the rewriting of the instruction behavior since pipeline registers will be changed too. For each pipeline stage, the designer defines a set of operations that are activated when the pipeline stage is reached in the simulation. This part of the description may be complicated. With HARMLESS, the designer assigns components (see section Hardware components description) to the pipeline stages to control the concurrence of components accesses. Components and instruction set descriptions are independent from the description of the pipeline and a change in the pipeline does not lead to a rewriting of the instruction set description or the component description. The HARMLESS compiler maps component accesses to the pipeline stages using devices automatically (see section Micro-architecture view). This way, the pipeline description is simpler to design and many pipeline descriptions may be done for the same instruction set and components description.

HARMLESS is based on 4 separate views: the format view, the syntax view, the behavior view and the micro-architecture view. A set of trees composes the 3 first views that do not have the same structure. The separation of these views enables the user to choose the best structure for every view. This approach allows to ease the description of an architecture in an incremental way. So, the description becomes more flexible, and easy to modify to re-target on another micro-architecture for example. The architectural scope is not limited, it allows the description of a wide variety of architectures. In addition, it supports multi-word instructions (variable length instructions), as well as unsigned and signed data type with any number of bits. At last, the micro-architecture view allows an easy description of a pipeline and changes in the micro-architecture view have no impact on the other views. The HARMLESS’s syntax is close to C language, so very easy to learn.

**Pipeline modeling**

The pipeline model in Harmless is done using an automaton. The goal is to shift a significative part of the computation from runtime to compile time (simulator generation). In this section, we focus on the key concepts that are used in automaton generation. Algorithms of the automation generation can be found in [Kassem et al. (2008)] and are implemented in the p2a tool which is a part of HARMLESS.

Sequential pipelines are considered in this paper (i.e. there are no pipelines working in parallel nor forking pipelines). The pipeline behavior is modeled in HARMLESS using an automaton, where a state represents the pipeline state at a particular time (see figure 1).

At each clock cycle, the pipeline goes from one state to another according to hazards. They are classified into three categories [Hennessy and Patterson (2001)]:

- **Structural hazards** are the result of a lack of hardware resources;
Data hazards are the result of a data dependency between instructions.

Control hazards that occur when a branch is taken in the program.

Control hazard are resolved in the simulator at runtime: instructions that are in the delay slot of a branch instruction are dynamically replaced by NOP instructions, if the branch is found to be taken.

Constraints resulting from structural and data hazards are used to generate this automaton and modeled using resources.

Resources

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

Two types of resources are defined, internal and external resources, that model respectively statically and dynamically constraints.

Internal resources can be compared to “resources” in [?]. They model structural hazards. As the pipeline state is known (i.e. the instructions are defined for each stage of the pipeline), then the state of each internal resource is fully defined (taken or available). In that case, when the automaton is built (and then the simulator), constraints described by internal resources are directly resolved when the set of next states is built.

As these resources are taken into account at build time (static approach), no computation overhead is required to check for this type of constraint at runtime.

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

External resources represent resources that are shared with other hardware components such as timers or memory controllers. It is an extension of internal resources to take into account resources that must be managed dynamically (i.e. during the simulation). For instance, in the case of a 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.

One interesting property of the external resources is that it allows to check for data hazards, as dependency between registers is resolved dynamically in function of the instruction flow.

Instruction class

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

The number of instruction classes is limited to $2^{R_{ext}+R_{int}}$ ($R_{ext}$ and $R_{int}$ are the number respectively of external and internal resources in the system), but this maximum is not reached because some resources are shared by all instructions, like pipeline stages, which leads to get a lower number of instruction classes.

The use of instruction classes is required to prevent a combinatorial explosion during the automaton generation.

Overview of the language

A HARMLESS description contains many parts:

- the instruction set (the set of operations that a processor can execute);
- the hardware components used by the instructions like memory, registers, ALU, 
- the micro-architecture (the pipeline);
- the peripherals like timers, input/output, 

Description of the instruction set

HARMLESS uses 3 independent views to describe the instruction set of a processor (more details may be found in [Kassem et al. (2009)]):

- the format view that describes the binary format of the instructions;
- the syntax view allows to describe the textual format of the instructions;
- the behavior view that binds a behavior to each instruction signature and provides a way to describe the components which are accessed by 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). A node description conforms to the following syntax.

```
<kind> <name> [#<tag>] <kind_options> {
  <description>
}
```

where <kind> can be format, syntax or behavior. As in a grammar specification language, HARMLESS allows to describe, for each view, whether a non-terminal node is built by aggregating sub-nodes or by selecting one node, or an aggregate of nodes, among several. By default, a non terminal node aggregates the sub-nodes. The select construct allows to choose one sub-node
among several (or an aggregate of sub-nodes among several).

Using this model, in each view, an instruction is represented by 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 a <tag> field. 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 may also appear in the <description> part of a node and indicate a leaf in the tree.

HARMLESS is a strongly typed language. It offers signed and unsigned data type with any number of bits. As a result, the description of the instruction set is more flexible and the designer may choose the best tree structure for each view.

Hardware components description

A component is a building block of the processor architecture like registers file, memory or ALU. In HARMLESS ADL, a component allows the functional description of a hardware component, this description is made in an object oriented way and contains variables as well as methods. A method allows an instruction to access a function offered by a component.

Let’s consider for example, a component named Fetcher that is in charge of the management of program counter, the description in HARMLESS is as follow (taken from the description of The Freescale XGate co-processor):

```c
1 component Fetcher {
2     program counter u16 pc; -- generate get and set methods.
3
4     void reset() {
5         pc := 0;
6     }
```

This component contains a variable member which is a register (here, a specific register called program counter pc), as well as 2 associated methods (reset and branch).

Furthermore, HARMLESS supports the description of a complex memory mapping (for detailed informations see [Kassem et al. (2009)]). The description is currently limited to the functional part and does not include any structural information (memory hierarchy, memory latency and alignment). However, this part is easy to add and it will be included in future works.

Micro-architecture view

The micro-architecture view allows to describe the micro-architecture that implements the instruction set of the processor. The goal is to map the instruction set behavior view on the micro-architecture view using the components.

The micro-architecture is described in architecture and pipeline sections. The architecture section forms the interface between a set of hardware components (like 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 devices to control the concurrency between instructions to access the same component.

Every device in the architecture is related to one component. The interest to differentiate the device from the component is to have the possibility of describing several devices to reach various manners to access the methods of one component.

Figure 1: A state of the automaton represents the state of the pipeline at a given time. In this example with a 4-stages 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 (this restriction is only made for clarity reason).
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 micro-architecture or shared (i.e. the port is not exclusively used by the micro-architecture).

The pipeline section allows to describe a pipeline. A pipeline is mapped on an architecture. In the pipeline description, all the stages are listed in order. In each stage the components and the methods that can be accessed by the instruction set are enumerated. They will be mapped on devices and ports of the architecture. Let’s consider for instance the following micro-architecture description given by two objects (architecture section and pipeline section) using the Freescale XGate instruction set ([Freescale 2003]).

The concurrency constraints are:

- the registers file is able to perform 3 reads and 2 writes in parallel;
- an Harvard architecture is used (separation between program and data memories);
- the computation in the ALU requires 2 stages and is not pipelined;

In HARMLESS ADL, the description of the first object (architecture) is as follow.

```java
architecture Generic {
  device gpr : GPR {
    read is read8 or read16
    write is write8 or write16
    port rs : read (3);
    port rd : write (2);
  }
  device alu : ALU {
    port all;
  }
  device MEM : mem {
    read is read8 or read16
    write is write8 or write16
    shared port fetch : read;
    shared port loadStore : read or write;
  }
  device fetcher : Fetcher {
    port branch : branch;
  }
}
```

In this description, an architecture named Generic with many devices is declared. For example, the device MEM 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 method read or write. The keyword or is equivalent to the exclusive or, and the two methods are aliases of methods declared in the component section.

The second object (pipeline) maps this architecture with 6 stages.

```java
pipeline sample_pipe maps to Generic {
  stage Fetch {
    mem : read;
  }
  stage Decode {
    Fetcher : branch;
    GPR : read;
  }
  stage Execute1 {
    ALU release in <Execute2> : *; /* means that any method of ALU can be used
  }
  stage Execute2 {
  }
  stage Memory {
    mem : read, write;
    GPR : read;
  }
  stage Register {
    GPR : write;
  }
}
```

In this example a pipeline, named sample_pipe, with 6 stages is declared and mapped on the Generic architecture. In this object, the components and their methods as well as the keywords (read and write) are used and will be mapped on the devices and ports of the architecture object during compile-time (see figure 2). The 6 stages of pipeline are listed:

- a Fetch stage to get the instruction from memory;
- a Decode stage which decodes the instruction, reads operands and performs branch instructions;
- 2 Execute stages;
- a Memory access stage;
- a Register stage that performs write accesses on the register bank.

When using a port in a pipeline stage, it is implicitly taken at the start of the stage and released at the end. If a port needs to be held for more than one stage, the
Figure 2: Mapping of the instruction set behavior view on the micro-architecture view using components access. On the left, methods of components accessed by the ADD instruction are shown. They are mapped on a 6-stage pipeline, using devices and ports that controls the concurrency between component accesses.

stage where it is released is explicitly given. Here port all of alu is taken in the Execute1 stage and released in the Execute2 stage.

Simulator generation

This section gives some details about how the simulator is automatically generated from the description. First, it introduces the model used for instructions. Then, how the micro-architecture description allows to generate the pipeline specification file (needed by the p2a tool) is explained, this file contains all the informations about the pipeline as the pipeline’s name, the stages names, the instruction classes and so on. Moreover, the generation of data dependency controller is presented.

Instruction Modeling

The simulator is generated in the C++ language. A C++ class models each instruction. in the HARMLESS description, an instruction corresponds to a path in the corresponding tree. The C++ class that represents an instruction offers 2 main methods:

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 stores values in the new object instance. It’s important to notice that the simulator context is never affected by this operation. This function is directly linked to the format description in HARMLESS.

The execution function
It is in charge to simulate the instruction. This function is directly linked to the behavior description in HARMLESS. Even, if the behavior description is split in multiple parts to take advantages of common behaviors, this function concatenates each part of the instruction behavior description, thus removing time consuming function calls in the generated simulator. Notice that, the instructions are executed independently of the progress of the pipeline (i.e. when an instruction is fetched, it is executed directly and after mapped on the pipeline stages).

Instruction classes generation

As each instruction depends on the fetch port and the pipeline stages, and the loadStore port is not used more than one time during one clock cycle (the access to memory can be only done in one stage), only 3 resources can differentiate instructions: there may be a maximum of $2^3 = 8$ instruction classes. From this description, we obtain 5 instruction classes. The 3 remaining instruction classes correspond to impossible configurations.

Data dependency Instruction classes generation

In the same way, based on the behavior view, the pipeline and architecture sections, instructions that use the same register ports (read or write a register) in the same stage of pipeline are grouped in data dependency instruction classes. Using the previous description of the micro-architecture, 5 data dependency instruction classes are obtained:
• a class that groups instructions that read a register in Decode stage;
• a class that groups instructions that read a register in Decode stage and write in a register in Register stage;
• a class that groups instructions that write in a register in Register stage;
• a class that groups instructions that read a register in Decode and Memory stages;
• a class that groups instructions that read a register in Decode and Memory stages and write in a register in Register stage.

Data dependency controller

During the simulation, to control the access to the registers, the data dependency controller is used and works as presented in algorithm 1. An external resource is associated to a data dependency controller. A data dependency occurs when an instruction needs to read a register that will be updated later in the pipeline, so this instruction stalls in the current stage until it has the access right to needed data register. Let’s consider the pipeline described previously and 2 instructions: ADD R1,R2,R3 in Fetch stage and SUB R2,R5,R6 in Decode stage. The ADD instruction needs R2 and R3 to enter in Decode stage, but the register R2 will be updated by the previous SUB instruction in Register stage so the request to the controller fails and the associated external resource is set to busy. The ADD instruction will be blocked in the Fetch stage until the SUB instruction writes the result in the Register stage.

Results

This section shows some results of the simulator build process and the computation time required to simulate a basic scenario. Special interest is ported to compare an ISS (with no temporal information) and different CAS that model 3 variations of a pipeline. The instruction set of the model is based on the Freescale XGate co-processor (16 bits RISC co-processor in the Star12X).

Results are available on table 1. In the CAS5 column, we model a basic 5-stage pipeline (Fetch, Decode, Execute, Memory and Write Back stages) (DLX pipeline, see Hennessy and Patterson (2001)). In column CAS6 and CAS7, we split the Execute stage in respectively 2 and 3 stages, making a 6-stage and a 7-stage pipeline. This is done with modifying a few lines in the pipeline section of the description, which shows the interest of splitting the instruction set description from the micro-architecture. We give an overview of simulator performances on a simple example that calculates a Fibonacci sequence.

Algorithm 1: Instructions and data dependency controller interaction during simulation.

\[
\text{if there is an instruction in the Fetch or Execute2 stage and the instruction will need operands in the Decode or Memory stages then} \\
\hspace{1em} - \text{The instruction sends a request to the controller;} \\
\hspace{1em} \hspace{1em} \text{if at least one register is busy then} \\
\hspace{1em} \hspace{1em} \hspace{1em} - \text{the request fails: the external resource associated is set to busy;} \\
\hspace{1em} \hspace{1em} \hspace{1em} \text{else} \hspace{1em} \hspace{1em} - \text{the request successes: the external resource associated is set to available;} \\
\hspace{1em} \hspace{1em} \text{else} \hspace{1em} \hspace{1em} - \text{the request successes: the external resource associated is set to available;} \\
\hspace{1em} \hspace{1em} \text{if there is an instruction in the Decode stage and the instruction will write a result in the Register stage then} \\
\hspace{1em} \hspace{1em} \hspace{1em} - \text{The instruction sends a request to the controller;} \\
\hspace{1em} \hspace{1em} \hspace{1em} \text{- The controller blocks the register that will be updated;} \\
\hspace{1em} \hspace{1em} \hspace{1em} \text{if there is an instruction in the Register stage and the instruction writes a result then} \\
\hspace{1em} \hspace{1em} \hspace{1em} \hspace{1em} - \text{The instruction sends a request to the controller;} \\
\hspace{1em} \hspace{1em} \hspace{1em} \hspace{1em} \hspace{1em} - \text{The controller unblocks the updated register;} \\
\]

<table>
<thead>
<tr>
<th>Description length (lines)</th>
<th>ISS</th>
<th>CAS5</th>
<th>CAS6</th>
<th>CAS7</th>
</tr>
</thead>
<tbody>
<tr>
<td>Time to generate ISS sources</td>
<td>0.43s</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Time to generate sources of pipeline model</td>
<td>-</td>
<td>0.6s</td>
<td>0.23s</td>
<td>0.89s</td>
</tr>
<tr>
<td>Number of states of the automaton</td>
<td>-</td>
<td>628</td>
<td>2341</td>
<td>8734</td>
</tr>
<tr>
<td>Number of transitions of the automaton</td>
<td>-</td>
<td>2614</td>
<td>10031</td>
<td>38274</td>
</tr>
<tr>
<td>Simulator source size (C++ lines)</td>
<td>13600</td>
<td>16000</td>
<td>18500</td>
<td>27500</td>
</tr>
<tr>
<td>Compilation time</td>
<td>12.1s</td>
<td>21.9s</td>
<td>22.1s</td>
<td>22.8s</td>
</tr>
<tr>
<td>Execution of 100 millions of instructions</td>
<td>4.80s</td>
<td>25.6s</td>
<td>27.6s</td>
<td>29.2s</td>
</tr>
<tr>
<td>Throughput (cycle/inst)</td>
<td>-</td>
<td>1.84</td>
<td>2.15</td>
<td>2.46</td>
</tr>
<tr>
<td>Execution of 100 millions of cycles</td>
<td>-</td>
<td>13.72s</td>
<td>13.42s</td>
<td>12.15s</td>
</tr>
</tbody>
</table>

Table 1: This table presents some results on the build process and the computation time of a simulator. ISS stands for Instruction Set Simulator (no timings) and CAS stands for Cycle Accurate Simulator. Timings are made on an Intel Core 2 Duo @ 2.4 GHz.

We can see that the simulator build process is quite fast: the simulator binary is built from the model in less than 25 s in all cases, even for a 7-stage pipeline.
which requires a big automaton. For very long pipeline models, we will have to split the pipeline model in two parts, to generate 2 automata.

The most important point concerns the simulation speed, indeed, the build process is performed only once. It refers to the last three lines of the table. The computation time required for the three CAS models are in the same order of magnitude. For a given number of instructions, the model with the shortest pipeline is the faster, but it also requires less cycles for the same job. We can notice that 25.6 s are required to simulate 100 millions instructions (184 millions of cycles). As a comparison, the ISS required 4.8 s for the same scenario, but without any temporal information. So the increase factor for computing timing properties is between 5.3 and 6.1, depending of the pipeline depth.

Another point of interest is that the execution of 100 millions of cycles is a little faster on the longest pipeline. This result can be explained by the fact that there is less instruction throughput on that model (data hazard penalty increases with pipeline depth). We can notice that the computation overhead induced by the addition of the pipeline stages is extremely low, due to the internal automaton approach.

Conclusion and ongoing work

This paper has described HARMLESS ADL. One of the goal of HARMLESS is to ease the description of an hardware architecture by providing separate views for the instruction set and the micro-architecture: binary format, assembly language instruction syntax, instruction behavior and micro-architecture. Splitting the description in views allows writing it in an incremental way and lowering the amount of work that is required to get a working description.

This paper specially focuses on how the micro-architecture view is organized, and how the high level model of the pipeline is translated to an automaton. The goal of the automaton model of the pipeline is to shift a significative computation part to compile time (simulator generation), to keep runtime simulation requirements acceptable.

Currently, a working HARMLESS compiler exists and generates both an instruction set simulator and a cycle accurate simulator, based on a sequential pipeline. The performance of the simulator is good and compares favorably to existing simulators. A prototype of HARMLESS compiler may be downloaded from http://harmless.rts-software.org.

Future work will focus on the minimisation of the automaton and the use of multiple automata to model and simulate superscalar processors. How to model dynamic superscalar processors, including speculative execution is also planned.

REFERENCES


