Instruction Set Simulator Generation Using HARMLESS, a New Hardware Architecture Description Language

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

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

ABSTRACT

Instruction set 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 a 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 introduces HARMLESS, a new language to automatically generate instruction set simulators. It differs from other 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 simulator. Some results are also presented on the simulator build process, especially on the decoder generation. An instruction cache is also introduced to speed up simulation in the same order of magnitude as hand-written simulators. Some experimental results are eventually presented.

Categories and Subject Descriptors

H.4 [**Information Systems Applications**]: Miscellaneous; D.2.8 [**Software Engineering**]: Metrics—*complexity measures, performance measures*

Keywords

Instruction Set Simulation, Hardware Architecture Description Language

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

SIMUTools 2009, Rome, Italy

1. INTRODUCTION

The interest of the simulation techniques for the design of software, in particular for embedded systems, is not to be demonstrated anymore. Simulation techniques don't 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.

Of course the simulator is the central element of this technique 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 on a fine grain model of the processor. Two simulation approaches are attractive. The Instruction Set Simulator (ISS) only takes into account the instruction behaviour, 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, a hardware Architecture Description Language (ADL) may be used.

The work presented in this paper is a part of a larger project aiming at the realization of a simulator offering ISS and CAS possibilities, and used for the design of real-time embedded systems. Some information about the CAS part are quickly given in this paper and the interested reader can refer to [5] for more details. This paper focuses on the automatic generation of an ISS. It introduces HARMLESS(Hardware ARchitecture Modeling Languague for Embedded Software Simulation), a new hardware Architecture Description Language (ADL). HARMLESS differentiates itself from other ADLs by using three independent views for binary format, syntax and behavior. This approach is more flexible and eases the description. Some internal aspects of the decoder generation and the execution approach are also shown.

The paper is organized as follows: Section 2 is dedicated to

Digital Object Identifier: 10.4108/ICST.SIMUTOOLS2009.5643
http://dx.doi.org/10.4108/ICST.SIMUTOOLS2009.5643

related work on other ISS generators. Section 3 introduces key features of HARMLESS description language. Section 4 highlights the benefits of the approach used in HARMLESS. In section 5, some internal aspects of this new ADL and some early results are given. Section 6 concludes the paper.

2. 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 have been done on ADLs that can be classified into three categories[6]: the first one that captures the Instruction Set (i.e., nML [2], ISDL [3]); the second one that captures the structure of the processor (i.e., MIMOLA [1]); and the third one that mixes the two first categories (IS and structure) ((i.e., LISA[7], EXPRESSION [4]). 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 the case of a CAS, it is necessary to add the description of the micro-architecture.

In [2], nML, an example of IS ADL, is presented. It is based on the standard description that can be found in the usual designer's manuals to specify the target architecture; this makes the language attractive. The architecture description contains structural/behavioral information. In nML, the Instruction Set (IS) of the target processor is described as an attributed grammar. This approach allows a hierarchical description of instructions. However, it describes the format (named image in nML), the behavior (named action in nML) and the syntax of instructions in the same view, as shown in this example:

```
op ADD(dest:REGISTER, src1:REGISTER, src2:ADDR)
syntax=format("add \%s,\%s,\%s,dest.syntax,
src1.syntax,src2.syntax)
image=format("00000000000\%s\%s\%s",dest.image,
src1.image,src2.image)
action= {dest = src1 + src2;}
```

The above code segment has action, image and syntax description for the 'add' operation. REGISTER and ADDR are addressing modes described elsewhere. The format operator, used for bit strings only, returns a string that can be interpreted as a number.

Therefore, the description of an architecture is not so easy to modify.

ISDL [3] is more flexible and its semantics are stronger than those of nML. It allows the description of a wide variety of architectures with emphasis on VLIW architectures. Like in nML, The IS contains behavioral and structural information. Moreover, it allows the description of the hardware of the micro-architecture, that is mixed with the behavior of the instructions.

The main goal of MIMOLA [1] is hardware-software co-design, this implies a much more 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 [7], S. Pees and al. presented LISA as a machine description language that gives a formal description of programmable architectures, their interfaces and peripherals. In many aspects, LISA includes ideas wich are analogue to nML, and the binary format, behavior and syntax of instructions are placed in the same view.

In the same way, EXPRESSION [4], a mixed ADL, gathers the binary format and the behavior of instructions in the same place.

None of the systems mentioned above provide a separate view for each IS description (format, behavior and syntax). On the contrary, to describe the instruction set of the processor, our language HARMLESS is based on 3 separate views: the format view, the syntax view and the behavior view. A set of trees composes each of these 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 (see section 4). In addition, it supports multi-word instructions (variable length instructions), as well as unsigned and signed data type with any number of bits. HARMLESS syntax is close to C language, so very easy to learn.

3. OVERVIEW OF THE LANGUAGE

HARMLESS uses 3 views to describe the instruction set of the processor. The first view deals with the format of the instructions, the second view allows to describe the syntax and the third view gives 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). 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. Since, it is targeted to instruction set description, it offers signed and unsigned data type with any number of bits. The language has some unusual features. For instance, the sum of two n bits words produces a $n + 1$ bits result. This way, the implementation of the carry or overflow computation is easier. Operators to extract and concatenate bit fields are provided:

```
u16 val1 := \x5500
```

Digital Object Identifier: 10.4108/ICST.SIMUTOOLS2009.5643
http://dx.doi.org/10.4108/ICST.SIMUTOOLS2009.5643

```
  u16 val2 := \x0055
  u17 result := val1+val2  -- result on 17 bits
  u1 carry := result{16}   -- only MSB
  u16 valResult := result{15..0}
```

3.1 The format view

The format view describes the binary format of the instructions.

A node in the format view gives, in the `<kind_options>` field, which part of the instruction word is scanned (this is called a slice in HARMLESS). For example, the following declaration taken from the Freescale HCS12 description scans 2 bytes. In the `<description>` part, a constant (offset) is extracted from these 2 bytes and is the concatenation of bit 4 of the first byte and the second byte.

```
format DIT_offset slice {7..0}+{7..0}
  offset := signed slice{4}{7..0}
end format
```

A select construct specifies a slice too. In the following example, the select chooses one of the nodes according to the value of the 3 lower bits.

```
format MOVW_ABA_DAA_inst
  select slice {2..0}
    case 6 is #ABA
    case 7 is #DAA
    others is MOVW_inst
  end select
end format
```

Here `ABA` and `DAA` are leaves in the tree. `MOVW_inst` refers to other nodes that specify the format of the addressing mode that `MOVW` instruction may use.

```
format MOVW_inst #MOVW
  select slice {3..0}
    case 3 is imm16_am ext_am
    case 0 is xb_am imm16_am
    ...
  end select
end format
```

The first addressing mode is shown where the source is a 16 bit immediate (`imm16_am`) and the destination an extended address (`ext_am`). In `imm16_am`, the `imm16_value` field is extracted:

```
format imm16_am slice +{7..0}{7..0} #IMM16
  imm16_value := signed slice{7..0}{7..0}
end format
```

A field may be signed or unsigned and has an implicit type that depends on the number of bits it uses. In the example above, `imm16_value` is a signed 16 bits integer (a s16) but any number of bits may be used. Bit masks (a binary number prefixed by `\m`) can be used to indicate which part of the slice is meaningless in the differentiation of the instructions. For instance, in:

```
    ...
    case \m111--00- is Indexed_9bits_offset
    ...
```

bits 0 (the offset sign), 3 and 4 (the register used) of the slice are used to decode the addressing mode (as denoted by the '-' in the mask). Some basic operations like left or right shifting and field concatenation may be performed when a field is extracted.

HARMLESS format description supports variable length instructions. In the example above, instructions `ABA` and `DAA` use 2 bytes. `MOVW` with a 16 bits immediate uses 2 more bytes as indicated by the '+' in `slice +{7..0}{7..0}`. 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. In some occasion, the same sub-format is used in more than one place. To differentiate same sub-formats, a suffix tag has to be added to the node name. For instance, in the HCS12 description, the indexed addressing mode may be used in both the source and the destination operands. This is indicated using the following description:

```
format idx_idx
  xb_am@SRC
  xb_am@DST
end format
```

where @SRC and @DST are the suffix tags.

3.2 The syntax view

The syntax view describes the textual format of the instructions. 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. For instance, the syntax for the HCS12 instruction `ABA` (this instruction signature has only one tag) is:

```
syntax ABA #ABA
  "ABA"
end syntax
```

The `MOVW` instruction syntax is more complex and uses other syntax nodes that are reused elsewhere:

```
syntax MOVW #MOVW
  "MOVW" mov16_args
end syntax

syntax mov16_args
  select
    case imm16 ext
    case imm16 idx
    ...
  end select
end syntax

syntax imm16 #IMM16
  field s16 imm16_value
  " #\d",imm16_value
end syntax
```

This description means that a `MOVW` instruction syntax is the concatenation of the string `"MOVW"`, a space, a `#`, the 16 bits immediate and so on. The keyword `field` allows to reference a field that is extracted from the instruction in the format view. The field must be typed (s16 here) and is checked against the format view.

Digital Object Identifier: 10.4108/ICST.SIMUTOOLS2009.5643
http://dx.doi.org/10.4108/ICST.SIMUTOOLS2009.5643

The <description> in a syntax node may use standard complex control structure like if... then... else. This is needed to give more flexibility in 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 set, like the PowerPC one, where the addi rD,0,value instruction translates to the li rD,value simplified syntax.

3.3 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. A component is a building block of the processor architecture like register file, memory or ALU. The description of components is made in an object oriented way and contains data as well as methods. For instance the following description shows a part of the alu component and one of its operations for the HCS12 model:

```
component alu {
  register u8 CCR {
    C := slice{0} -- carry flag
    V := slice{1} -- overflow flag
    Z := slice{2} -- zero flag
    N := slice{3} -- neg flag
    I := slice{4} -- maskable interrupt
    H := slice{5} -- half carry status bit
    X := slice{6} -- non-maskable interrupt
    S := slice{7} -- STOP instruction
  }

  u8 neg_8(u8 op) {
    u8 res;
    res := 0 - op;
    CCR.N := res{7};
    CCR.Z := res = 0;
    CCR.V := op = \x80;
    CCR.C := op != 0;
    return res;
  }
  ...
}
```

As for the other 2 views, the remaining of the behavior view is a set of nodes which describe a piece of behavior. A behavior node contains a declaration section with local variable declarations, other behavior node references and one or more do blocks to specify the algorithm of the instruction. The following description presents the NEG instruction example among other:

```
behavior mono_operation8(u8 op, out u8 res)
  select
    case #COM  do res := alu.com_8(op);     end do
    case #ASL  do res := alu.sl_8(op);      end do
    case #ASR  do res := alu.asr_8(op);     end do
    case #DEC  do res := alu.add_8(op,0-1); end do
    case #INC  do res := alu.add_8(op,1);   end do
    case #NEG  do res := alu.neg_8(op);     end do
    case #ROL  do res := alu.rol_8(op);     end do
    case #ROR  do res := alu.ror_8(op);     end do
    case #LSR  do res := alu.lsr_8(op);     end do
  end select
end behavior
```

```
behavior monadic8_reg_inst #INH_AM
  u8 reg;
  u8 res;
  get_reg8(reg)
  mono_operation8(reg,res)
  put_reg8(res)
end behavior
```

Here, the monadic8_reg_inst behavior declares 2 local variables to store the 8 bits register content (source) and the result of the operation. Then it gets the content of the 8 bits register from the get_reg8 behavior, performs the operations with the mono_operation8 behavior and stores the result using the put_reg8 behavior. In the mono_operation8 behavior, one of the operations is selected. Parameters may be passed from one behavior to another by reference (using the out keyword) or by value.

The HCS12 description, used to illustrate this section, is quite long. It takes 2500 lines of which 28% for format view, 10% for syntax view and 62% for behavior view (20% for components and 42% for the tree nodes).

3.4 Memory description

HARMLESS supports the description of a complex memory mapping. The description is currently limited to the functional part and does not include any structural information (memory hierarchy, memory latency and alignment). The structural part is used to extract the time required for a memory access and will be included in future works to generate a Cycle-Accurate Simulator.

The memory description is embedded into a component (see section 3.3). Standard parameters are given to a memory chunk, such as the bus width, address range or the type of access (RAM, ROM or register):

```
component mem {
  program memory internalRam {
    width   := 16 -- get 16 bits / access max
    address := \x0..\xFFFF
    type    := RAM
  }
```

The memory can be read or written (not for ROM memory) in other components or in the instruction behavior parts using implicit methods: u16 mem.read16(u32 addr), void mem.write16(u32 addr, u16 data) in this example (16 bits). For convenience, a read8 and write8 methods are also generated.

The program keyword is used to determine memory that can accept a program code at startup. Obviously, there should not be 2 memory zones with the same address using the program keyword.

3.4.1 Alias

An alias represents a reference to a memory location, that could be accessed in a more convenient way. It is useful for registers that could be accessed using their names, and other memory chunks. HARMLESS allows to map a memory chunk relatively to an expression. For instance, let's consider the Infineon C166 internal RAM :General Purpose Registers are mapped in the internal RAM, based on the value of the Context Pointer (CP). This may be described as follows:

Digital Object Identifier: 10.4108/ICST.SIMUTOOLS2009.5643
http://dx.doi.org/10.4108/ICST.SIMUTOOLS2009.5643

```
component mem
  memory ram {
    address := 0...\xFFFF
    type    := RAM
    width   := 16

    -- register alias. Size is 16 bits
    register CP maps to \xFE10

    -- GPR are mapped in memory relatively to CP
    GPR {
      type := register
      address := 0..15
    } maps to CP
  }
}
```

Using this example, some new methods relative to the mapping are generated: u16 mem.ram.GPR(u16 addr) will read the value of CP, add the address value in parameter and perform a read access in the memory. Registers can be used like a classic register defined everywhere in the description (*i.e. CP := newVal*).

3.4.2 Adding user defined methods

Embedding the memory description into a component allows to add user methods to facilitate the memory access (paged, segmented memory). Here is a partial description of the HCS12 memory mapping, using register PPAGE for the program page (flash)[1]:

```
component Mem {
  memory registers {
    width := 16
    address := \x0..\x7FF -- 2 kb of registers
    type := register

    register u8 PPAGE maps to \x30
  }
  program memory windowedFlash {
    width := 16
    address := \x78_FFFF..\x7F_FFFF -- 512 kb
    type := ROM
  }
  u8 memRead8(u16 addr) {
    -- description from MC9S12XDP512, p.31
    u8 val := 0
    if     addr < \x0800 then
      val := Mem.registers.read8(addr)
    elseif addr > \x8000 & addr < \xC000 then
      u24 ra := 1 cat PPAGE{7..0} cat addr{13..0}
      val := Mem.windowedFlash.read8(ra)
    end if
    return val
  }
}
```

3.5 Micro-architecture view

The micro-architecture view allows to describe the micro-architecture that implements the instruction set. The goal is to map the instruction set behavior view on the micro-architecture view using the components. In this way, a cycle accurate simulator can be generated. This simulator can be used to verify timing characteristics of embedded real-time systems. The execution model of this simulator is a finite state automaton generated from the description of the pipeline in HARMLESS. More details may be found in [5]

A state of the automaton represents the pipeline state at a particular time. At each clock cycle, the pipeline goes from one state to another according to hazards. They are classified into three categories:

- *Structural hazards* are the result of a lack of hardware resource;
- *Data hazards* are the result of a data dependency between instructions;
- *Control hazards* that occur when a branch is taken in the program.

To illustrate how a pipeline can be described in HARMLESS, let's consider an example with a 2-stage pipeline, with only 1 instruction (Nop), 1 component (the Memory) and one temporal constraint (the memory access in the fetch stage). Using the HARMLESS ADL, this pipeline can be described as follows:

```
architecture Generic {
   device mem : Memory {
      shared port fetch : getValue;
  }
}

pipeline pFE  maps to Generic {
   stage F {
       mem : fetch;
   }
   stage E {
   }
}
```

In the description above, two objects are declared:

- *T*he `architecture` named `Generic` that describes the micro-architecture;
- *T*he `pipeline` named `pFE`.

The `architecture` section forms the interface between a set of hardware components and the definition of the pipeline. It allows to express hardware constraints having consequences on the temporal sequence of the simulator. The architecture contains one device (`mem`) to control the concurrency to access the Memory component.

In this description: at a given time, the method `getValue`, that gets the instruction code from the memory, can be accessed one time using the `fetch` port. We suppose this access can be made concurrently by other bus masters. So the port is `shared`.

The pipeline `pFE` is mapped to the `Generic` architecture. The 2 stages of the pipeline are listed. In stage `F`, an instruction can use the `fetch` port.

This description leads to generate a 4-state automaton (detailed in [5]). The experimentation shows good performance: the cycle accurate simulator is less than 4 times slower than the ISS. Moreover, the design of the cycle accurate simulator allows to disconnect dynamically the micro-architecture simulation to run at the speed of the ISS until interesting program sections are reached.

[1]The `cat` operator is used for concatenation.

Digital Object Identifier: 10.4108/ICST.SIMUTOOLS2009.5643
http://dx.doi.org/10.4108/ICST.SIMUTOOLS2009.5643

4. USEFULNESS OF A 3 VIEWS DESCRIPTION

The main difference between HARMLESS and the other HADL is the way the description is handled. In the formers, the description is monolithic with the binary format (named image), the syntax and the behavior (named action in nML) sharing the same description tree. In the latter, each view has its own tree. This way the designer may choose the best tree structure for each view.

To illustrate the advantage of the HARMLESS description, let us present two examples taken from the HCS12 description. Figure 1 shows the sub-tree of the format view of the CALL instruction family. Boxed shapes are the tags, vertical bars are the aggregates and round shapes are the alternates. As explained in section 3, a branch in this tree corresponds to an instruction of the CALL family and is defined by the set of tags in this branch. Binary masks used to decode the instructions and to build the view are shown above the boxed shapes.

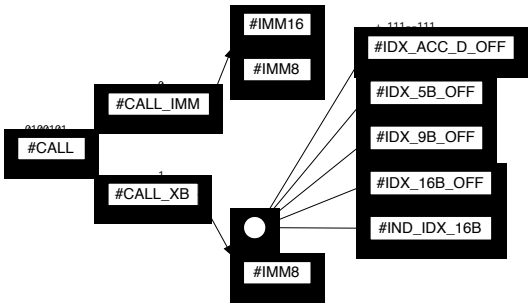

Figure 1: Format sub-tree of the CALL instruction family

Figure 2 shows the behavior view of the CALL with immediate address instruction and CALL indirect instruction family. This time the view has been built with the behavior in mind and the trees reflect the semantic differences.

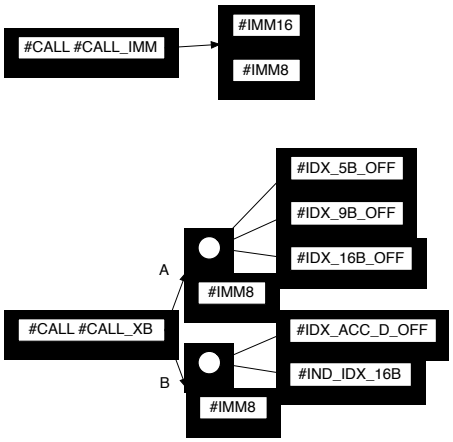

Figure 2: Behavior sub-trees of the CALL with immediate address instruction and CALL indexed or indirect family

As expected, there is not a direct correspondence between the format tree and the behavior tree. In the behavior tree shown in figure 2, the branch labeled 'A' corresponds to CALL with the indexed addressing modes (the target address is computed by summing the content of a register with a constant) while the branch labeled 'B' corresponds to CALL with the indirect indexed addressing mode (the target address is read from memory at an address computed by summing register D and a constant or given as an immediate value in the instruction).

Another advantage of this description is the ability to generalize the behavior description. For instance, most processors do not have an orthogonal instruction set (all instructions do not operate on all the registers) and it is tedious to describe every register access variant in the behavior of each instruction. So one can describe the behavior by assuming all registers are accessible. When the description is compiled, HARMLESS removes all the behaviors that do not have a corresponding format. This is shown in the example below.

Figure 3 shows the format tree of the TFR (transfer) instruction. This instruction moves data from a source to a destination registers. Register TMP2 may only be a destination while register TMP3 may only be a source.

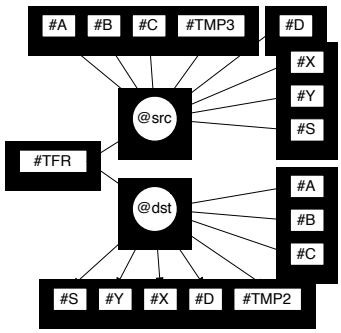

Figure 3: Format tree of the TFR instruction

Figure 4 shows the behavior tree of the 16 bits TFR instruction (both source and destination are 16 bits registres, TFR to/from 8 bits registers exist too).

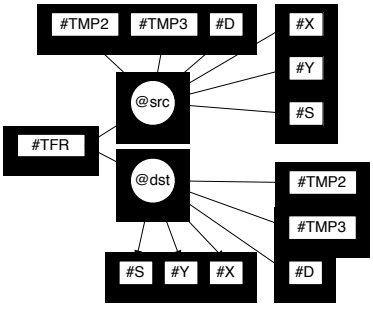

Figure 4: Behavior tree of the TFR instruction

The behavior describes instructions that do not exist (all the TFR with TMP2 as source and all with TMP3 as destination) but since the format for such instructions does not exist, the simulator generated by HARMLESS does not include these instructions.

Digital Object Identifier: 10.4108/ICST.SIMUTOOLS2009.5643
http://dx.doi.org/10.4108/ICST.SIMUTOOLS2009.5643

These features allow to simplify the description. As a result, the nML HCS12 description takes 7400 lines while the HARMLESS takes 2700 lines.

5. 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, the decoder generation is explained. Moreover, an improvment is presented using an internal cache, in order to speed up the simulation process. Finally, some results on the generation process are given on three processor descriptions.

5.1 Instruction Modeling

The simulator is generated in the C++ language, and each instruction is modeled using a C++ class. An instruction, in the HARMLESS description, corresponds to a path in the corresponding tree. The C++ class that represents an instruction offers 3 main methods:

5.1.1 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. For example, consider an addition instruction on a RISC architecture: ADD R1, R2, R3. The constructor extracts, from the binary code of the instruction, the index of the destination register (1), and the indices of the 2 source registers (2) and (3). 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.

5.1.2 The execution function

It is in charge of simulating the instruction. Using the previous example based on the addition instruction, this function will read the value of the 2 source registers (the register index is known), perform the addition and update the flag register, and finally write back the result in the destination register. 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. If no behavior description is given for an instruction, a default one is used that warns the user when executed. This is helpful when used in an incremental description approach.

5.1.3 The mnemonic function

It is used for disassembling. This function does not modify the simulator context. The function is associated to the syntax description in HARMLESS. Anyway, if no syntax description is given for an instruction, a default one is used, returning the internal name of the instruction.

5.1.4 Classical interpretive execution approach

The execution of an instruction is initially based on an interpretive approach, executing each instruction one after the other. The execution process is given in figure 5. First, the decoder phase has to decode the binary code pointed by the instruction pointer (explained in depth in section 5.2). Then, the instruction object is created (requiring a memory allocation), the execution of the instruction is performed and the instruction object is deleted (requiring a memory deallocation). This phase has to be processed for each instruction in the program, and memory allocation/deallocation are particularly penalizing in computation time. A more efficient approach is explained in section 5.3.

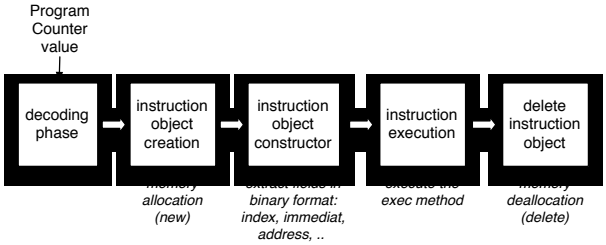

Figure 5: Basic execution of an instruction

5.2 Decoder generation

Based on the format description part, the decoder is a key 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, for each instruction, all the format parts used in the description. To each format, a couple mask/value allows to identify bits that should be positioned (0 or 1) and bits that are not representative. Let's consider, for instance, the first example in section 3.1. For all the instructions, that contain the `TBL_inst` node, the mask/value will be `0xF/0xD`. This is a condition that means: only the four lowest significant bits are taken into account (mask is `0xF`), and the value gives the state of these 4 bits. The final code of the instruction is represented by the conjunction of all the conditions (mask/value) related to each format part.

In order to facilitate treatments, the internal representation of conditions (mask/value) is encoded using Binary Decision Diagrams (BDD). This allows to verify very simply the orthogonality of the instruction set. If two instructions may have the same binary code, then the conjunction of their BDD is not an empty BDD. It's sufficient to make the conjunction of the BDD to all combinations of two instructions.

With the internal use of BDD, we obtain simple conditions, independently of the underlying description. The condition is applied on the instruction binary code pointed by the program counter. Here is an example with 2 conditions that can match for an instruction, but most instructions are decoded using only one condition:

```
if(((code & mask1) == value1) ||
    (code & mask2) == value2)) {
        //instruction found
}
```

The decoder could be directly generated using a list of conditions like this one, for each different instruction. The size of the masks and values is independent of the instruction code size and is defined to minimize the treatment on the host machine (32 bits). On the HCS12 for instance, instruction size is set between 1 and 8 bytes. If the instruction size is lower than 5 bytes, only 1 simple comparison is made.

Digital Object Identifier: 10.4108/ICST.SIMUTOOLS2009.5643
http://dx.doi.org/10.4108/ICST.SIMUTOOLS2009.5643

Then, a decoder can be generated in only one big function that embeds all comparisons. If the comparison related to the instruction that have to be decoded is at the end of this big function, many tests will be required to decode the instruction. In order to minimize the number of comparisons, we made the assumption that in most instruction sets, a part of the binary code used for the decoder (no register index, nor immediate..) is defined in the bits of the binary code. Then, based on the first byte of the instruction code, $2^8 = 256$ functions are generated to decode the rest of the instruction. This assumption leads to minimize significantly the number of comparisons required in the examples studied (Atmel AVR, Freescale HCS12, XGate, PowerPC, and Microchip PIC). The number of bits used to generate sub-decoder function has been set to 8 by default but can be modified. For instance, with the Microchip PIC10 (instruction size is 12 bits), the value can be increased to 12, to obtain $2^{12} = 4096$ sub-decoder functions with only one comparison.

5.3 Adding an instruction cache

The execution process of the classical interpretive execution approach, presented in figure 5, has two major structural drawbacks:

- when simulating an instruction inside a loop, the instruction will be decoded many times. It does not take into account the temporal locality;
- memory allocation/deallocation requires most of the computation time, when creating and deleting the C++ instruction object;

To enhance the previous approach, we add an instruction cache during the decoding phase. This approach is a simpler version of the *Instruction Set Compiled Simulation* in [8]. The instruction cache is internal, it has only low side effects (program memory should not be modified during simulation) and keeps all the advantages related to interpretive simulation approaches.

The simulation cache principle is described in figure 6. The first time the instruction is decoded, the instruction cache returns a miss and an instruction C++ object is allocated as in the previous approach. The new instruction is stored in the cache and is not deleted (but considering the replacement policy, an instruction object may be deallocated). The next time the instruction should be executed, the instruction object is in the cache and many time consuming phases may be removed: the decoding phase, the object allocation and the object constructor.

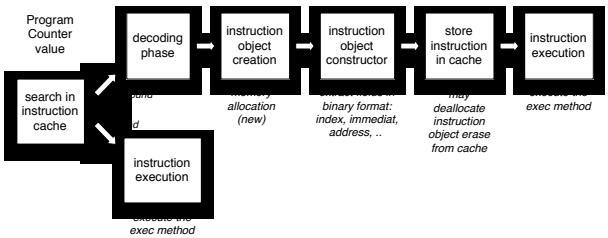

Figure 6: Execution of an instruction using the internal instruction cache.

The internal cache used is a direct-mapped one. This is the fastest to run in software. We compared it with a 2-way set-associative cache with an LRU replacement policy, but the computation overhead of the latter did not compensate the better hit ratio on the sample programs used.

5.4 Results

This section shows some results on the generation process of different processors: The HCS12, which is a CISC with a variable size instruction set, from 1 to 8 bytes; The Freescale XGate co-processor for the HCS12. It is a 16 bits RISC co-processor (instruction length is 16 bits); The Atmel AVR, which is a 8 bits RISC processor (instruction length is 16 bits), even if few instructions use 32 bits.

These results are available on table 1. We simulate one simple example on each processor, based on calculating a Fibonacci sequence. This gives an overview of simulator performances.

	HCS12	**XGate**	**AVR**
description length (lines)	2580	1150	1140
instructions generated (nb)	5517	88	87
Time to generate the simulator source from HARMLESS description (s)	20.7s	0.31s	0.27s
simulator source size (C++ lines)	~ 304 000	~ 8 200	~ 8 700
Time to compile the simulator (s)	215.85s	7.30s	7.72s
Time to execute 100 millions of instructions of the basic example (s)	4.06s	4.55s	4.92s

Table 1: This table presents some results on the generation of a simulator. Timings are made on an Intel Core 2 Duo @ 2.4 GHz.

We can see that RISC based architectures have few instructions, whereas the CISC one get more than 5500. This is not only due to the important number of addressing modes, but also due to the HCS12 architecture. There are only few registers on the HCS12, and for instance, the instruction that rotates left is expanded in two instructions `ROLA` and `ROLB`, depending on the register considered (`A` or `B`). In the description, we can either describe one instruction ROLx with one field parameter, or describe two different instructions. We chose the later to take advantages of the internal instruction cache: increasing the decoder complexity and simplifying the behavior description.

As the HCS12 model has many instructions, it leads to increase the time to generate the simulator, and the time to compile the generated C++ files. We can notice that if we remove the instruction set orthogonality check (that is relevant only when describing the format part), the time to generate simulator sources falls to 11.7s. An ISS simulator is built in less than 4 minutes from the description to the simulator binary application.

Digital Object Identifier: 10.4108/ICST.SIMUTOOLS2009.5643
http://dx.doi.org/10.4108/ICST.SIMUTOOLS2009.5643

6. CONCLUSION AND ONGOING WORK

This paper has described HARMLESS ADL. One of the goals of HARMLESS is to ease the description of a hardware architecture by providing separate views for binary format, assembly language instruction syntax and instruction behavior. This approach proved that it was attractive. 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. A 3-view description is more flexible and can describe efficiently various instruction set architectures. As mentioned above, the nML HCS12 description takes 7400 lines while the HARMLESS takes 2700 lines.

Currently, a working HARMLESS compiler exists and generates an instruction set simulator from the IS description. The performance of the simulator is good and compare favorably to existing ISS. A prototype of HARMLESS compiler may be downloaded from:
`http://p2a.rts-software.org`.

A fourth view (introduced in 3.5) is currently under development. It allows to describe the micro-architecture. The execution model of this simulator is a finite state automaton. Future work will focus on the minimisation of the automaton, the use of multiple automata to model and simulate superscalar processors. How to model dynamic superscalar processors, including speculative execution is also planned.

7. REFERENCES

[1] Steven Bashford, Ulrich Bieker, Berthold Harking, Rainer Leupers, Peter Marwedel, Andreas Neumann, and Dietmar Voggenauer. The mimola language version 4.1. Technical report, Lehrstuhl Informatik XII University of Dortmund, Dortmund, 1994.

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

[3] George Hadjiyiannis, Silvina Hanono, and Srinivas Devadas. Isdl: an instruction set description language for retargetability. In *DAC '97: Proceedings of the 34th annual conference on Design automation*, pages 299–302, New York, NY, USA, 1997. ACM.

[4] A. Halambi, P. Grun, and al. Expression: A language for architecture exploration through compiler/simulator retargetability. In *European Conference on Design, Automation and Test (DATE)*, March 1999.

[5] Rola Kassem, Mikaël Briday, Jean-Luc Béchennec, Yvon Trinquet, and Guillaume Savaton. Simulator generation using an automaton based pipeline model for timing analysis. *IMCSIT, International Workshop on Real Time Software (RTS'08), Wisla, Poland*, 2008.

[6] Prabhat Mishra and Nikil Dutt. Architecture description languages for programmable embedded systems. In *IEE Proceedings on Computers and Digital Techniques (CDT), Special issue on Embedded Microelectronic Systems: Status and Trends*, volume 152, pages 285–297, May 2005.

[7] Stefan Pees, Andreas Hoffmann, Vojin Zivojnovic, and Heinrich Meyr. 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*, pages 933–938, New York, NY, USA, 1999. ACM.

[8] Mehrdad Reshadi, Prabhat Mishra, and Nikil Dutt. Instruction set compiled simulation: A technique for fast and flexible instruction set simulation. In *DAC '03: Proceedings of the Conference on Design Automation*, 2003.

Digital Object Identifier: 10.4108/ICST.SIMUTOOLS2009.5643
http://dx.doi.org/10.4108/ICST.SIMUTOOLS2009.5643