Available online at www.sciencedirect.com

ScienceDirect

Electronic Notes in Theoretical Computer Science

Electronic Notes in Theoretical Computer Science 304 (2014) 111–125

www.elsevier.com/locate/entcs

K Semantics for Assembly Languages: A Case Study

Mihail Asăvoae

Computer Science Department
University "Al I Cuza"
Iasi, Romania

Abstract

Formal verification of embedded software systems often requires a low-level representation of the program under scrutiny. It is often the case that the verification tools rely on ad-hoc encodings of particular assembly language semantics. In this paper we use the $\mathbb{K}$ framework to formally define a MIPS-based assembly language. Our proposed definition is modular in the sense that it accommodates various organizations for the storage-related aspects of the semantics. We also present how to instantiate our $\mathbb{K}$ language definition on two main memory models, corresponding to different representations of the assembly code. Such a formal language definition could be directly used by the program verification tools.

Keywords: formal executable semantics, assembly language, the K framework

1 Introduction

Reasoning about behavior of embedded systems often requires knowledge about a program execution on a particular architecture. Thus, it is important to bridge the gap between these two components and to consider a convenient, low-level representation of the program. The assembly level is such a representation as it offers information about registers, instruction memory addresses, code interaction with the memory system (via load/store instructions), information about data placement etc.

Prominent areas of research on formal methods applied at the assembly-level code include: software security with detection and analyses of malicious behavior, compiler design with correct-by-construction code generation and embedded systems with non-functional requirements guarantees of the executables. The latter category includes the problem of formally deriving execution timing bounds for a program,

[1] Contract ANCS POS CCE, O2.1.2, ID nr 602/12516, ctr.nr 161/15.06.2010, DAK.

[2] European Social Fund in Romania, under the responsibility of the Managing Authority for the Sectoral Operational Programme for Human Resources Development 2007-2013 [grant POSDRU/88/1.5/S/47646]

[3] Email: mihail.asavoae@info.uaic.ro

http://dx.doi.org/10.1016/j.entcs.2014.05.006

with respect to the underlying architecture. It is required that the assembly code representation presents certain guarantees (i.e. absence of run-time errors). It is common to employ static analysis techniques on the high-level code and to assume that the compilation preserves the original properties of it. An alternative is to apply the analyses directly on the assembly code. We adhere to the latter alternative, from the following perspective: define the language semantics and use it to develop analysis tools.

In this paper, we use $\mathbb{K}$, a rewrite-based framework for programming languages design and analysis, to give a formal executable semantics of a RISC assembly language. We opt for the Simplescalar toolset PISA assembly language, in [2], which is based on the MIPS IV language. This toolset comes with a modified `gcc` compiler to generate such assembly code and eases the experimentation with various architecture features (i.e. caches, prediction mechanisms etc). While the latter point is beyond the scope of this work, the former helps in testing the semantics, via automated assembly code generation.

A first desiderate of our formal executable semantics of the RISC assembly language is the following - *one rewrite rule for each language instruction.* In this way we achieve a two-layered organization of the definition: the first layer contains the transformation of each instruction semantics into support operations and the second layer contains the implementations of these operations. For example, an ALU instruction is transformed into error checks, if necessary, followed by the corresponding source registers look-ups and the destination register update.

A second desiderate of our $\mathbb{K}$ executable semantics is to accommodate a uniform definition of semantic rules applicable to different representations of an assembly program. According to this, the most important design decision is to split the language configuration into the semantic entities corresponding to the memory configuration and the register file, the former getting two instances and the latter remaining unmodified. This requires a communication interface to facilitate the interaction between these two. Another important design decision is to define a specialized module for the support operations of the semantics, signed and unsigned integers as well as single and double precision floating point.

We distinguish between the following two representations of an assembly language program: (1) the assembly code obtained directly from a high-level program, say a C program and (2) a representation of the disassembled executable. Fig. 1 displays a simple C function that computes the sum of elements in an array in the left, the encoding of (1) in the middle and (2) in the right.

There are several distinctive features between these two variants. The direct encoding of the assembly program, in Fig. 1 (middle), uses labels as an abstract representation of the memory addresses for both instructions (i.e. `L1`, `L2`) and data (i.e. the array `a`). This makes it suitable for constructing the initial configuration of the program, as for example, the array `a` is initialized in the `.rdata` section. The representation of a disassembled executable provides exact information about the code memory layout (i.e. the concrete addresses 1000, 1004 etc), but makes it difficult to construct the initial program state. Our implementation is capable to handle both

formats, provided that certain requirements are satisfied. We elaborate on these in the implementation section.

Related Work. Developing formal methods for assembly code is important for a wide range of applications. In computer security, there are techniques for detection of malicious code behavior [5] and protection against virus infection using code obfuscation [17]. In compiler design, the current approaches follow the principles stated in [14], either to generate correct-by-construction assembly code [16] or assembly code extensions, such as a type system [22], to facilitate its validation. Another important application is the worst-case execution time (WCET) analysis of embedded code, when it is necessary to guarantee execution time bounds for a particular program. For a survey on timing analyses, we refer the reader to [28]. All the previously mentioned applications require, usually ad-hoc, implementations of assembly language semantics. Our approach is to provide a formal executable semantics, which should serve as a basis for developing analysis methods. We use the $\mathbb{K}$ framework, a specialization of rewriting logic [21,18].

In the same class of formal executable semantics of assembly languages there are the theorem-proving based approaches. One of the first works uses the ACL2 [15] prover to certify microcode programs for the Motorola CAP DSP [3]. The processor is modeled at both the instruction set and the pipeline levels. Another similar formalization is of ARM6 in [9] and ARM7 in [10], both using HOL theorem prover [11]. All these assembly language encodings are in the context of formal verification of the underlying architecture, and embeds correctness criteria with respect to the properties of interest. Our proposed $\mathbb{K}$ language definition focuses on a different aspect, it is designed to be more modular to accommodate various formats of the input program, without changing the semantics.

The Maude system [6,7] is the implementation of rewriting logic and together with a number of integrated methodologies and tools, it facilitates the design and analysis of systems. To the best of our knowledge, there are two approaches to model assembly languages. The work in [12] proposes a first model of a simplified RISC assembly language, used in verification of various microprocessor elements. The method presented in [27] presents a limited subset of the x86 assembly language for malware behavior detection. Both of these approaches focus more on verification issues and less on the language definition ones. Language semantics definitions rely on the memory system specification, both at the structural and functional levels. With respect to the memory representation in formal language definitions using rewriting logic, our current work complements the work in [13], which proposes memory models for imperative and object-oriented languages.

The $\mathbb{K}$ framework, described in [26], is a rewrite based framework that enables the definitions of operational semantics of programming languages. $\mathbb{K}$ shows its versatily when handling definitions of real languages, such as C in [8], Scheme in [19] and Verilog in [20]. In our previous work [1], we succinctly introduced the integer subset of the language of interest. The current implementation of $\mathbb{K}$ is implemented on top of the Maude system and is called $\mathbb{K}$-Maude. In this way, it has access to all integrated technologies and tools that Maude offers.

Outline of the paper. The paper is organized in the following way. Section 2 briefly overviews some concepts of the $\mathbb{K}$ framework as we formally define the assembly language. Section 3 describes a modular system with an emphasis on the main memory modeling and two of its possible instances. Section 4 presents implementation and benchmarking specific details, while Section 5 contains the conclusions.

2 Assembly Language in $\mathbb{K}$

The $\mathbb{K}$ framework facilitates definitions of modular and executable programming language semantics, using a specialized, concise notation to represent and manipulate language configurations. A language definition has three components: the language syntax, the configuration and the language semantics. The (abstract) syntax in $\mathbb{K}$ uses the BNF form and could include annotations with respect to the evaluation order. A language configuration represents the set of necessary semantic entities to capture the programming language semantics. In $\mathbb{K}$ a language configuration is represented as a multiset of cells. A cell, written as $\langle cont \rangle_{\mathsf{lbl}}$ is identified by the cell content *cont* (also a multiset of cells) and the cell name (or label), lbl.

The rules in $\mathbb{K}$ are of two kinds: *computational rules*, which are transitions in a program execution, and *structural rules*, that make a term amenable to the application of a computational rule. The $\mathbb{K}$ framework distinguishes between structural and computational rules as a mechanism to control the abstraction degree of a definition. This split follows the rewriting logic approach, where a rewrite logic theory is represented as a transition system with the states represented by equations (i.e. structural rule in $\mathbb{K}$) and the transitions represented by rewrite rules (i.e. computational rule in $\mathbb{K}$). In the $\mathbb{K}$ framework, a rule is by default computational. $\mathbb{K}$ uses a generalization of the usual rewrite rule, in the sense that a rewrite rule could also manipulate parts of a rewrite term (as read, write or don't care). The compact, bi-directional, notation represents the lefthand side of a rule placed above a horizontal line and the righthand side placed below. For a more elaborate presentation on the framework and its current implementation, we refer the reader to [26] and respectively [25].

We give more practical insights into $\mathbb{K}$ when we define the Simplescalar [4] PISA assembly language, which we call *SSRISC*. Next, we present a subset of it that includes integer and floating point ALU-instructions, branch and jump instructions, load and store instructions, an instruction for program errors. In other words, the subset exposes many of the language semantic entities, using the $\mathbb{K}$ specialized notation.

Apart from the instruction set presented in [4], there is a set of instructions that appear in the direct encoding of the assembly language (i.e. instructions `la` or `move`) or a number of pseudo-instructions that appears in the executables. Both kind of instructions have been defined as well. For presentation purposes, we include only the instruction `la` and omit to describe all the pseudo-instructions. The general methodology for language definitions in $\mathbb{K}$ is standard: first, we define the (abstract) syntax, then we present the language semantics, using transformations on

```
main (void) {
   int a[3]={-2,1,4};
   int i, sum=0;
   for (i=0; i<3; i++)
         sum=sum+a[i];

   return sum;
}
```

```
      .rdata
a:  .word -2
      .  .  .
      .text
      la $3,a
      lw $4,0($3)
      .  .  .
L1: lw $2,32($fp)
      slt $3,$2($3),3
      bne $3,$0,L2
      j L3
L2: .  .  .
L3: .  .  .
```

```
                .  .  .
1000 lui $3,4096
1004 lw $4,0($3)

1028 lw $2,32($30)
1032 slti $3,$2,3
1036 bne $3,$0,1044
1040 j 1060
1044 .  .  .
1060 .  .  .
```

Fig. 1. C program (left) with snapshots of direct assembly code (middle) and disassembled code (right)

the structural elements that make the configuration.

The *SSRISC* definition is designed to be concise and modular. From an organizational point of view, the formal executable semantics presents two layers: an structural layer in which each instruction is transformed into simpler operations and a computational layer containing the implementation of these particular support operations (i.e. register look-up, register update, program counter implementation). With the structural layer, our definition has one $\mathbb{K}$ rewrite rule per language instruction, making it amenable for extensions.

2.1 Syntax

The annotated $\mathbb{K}$ syntax for the subset of *SSRISC* assembly language is in Fig. 2. In the square brackets there is a special $\mathbb{K}$ notation, called strictness attribute. The keyword `strict` means that the particular operand on which strictness apply is reduced to a base value, called *KResult*. For example, the `add` instruction is *strict* on the second and third operands, which implies that the last two registers, called sources, are reduced to values before the actual addition takes place and the first operand, the destination register, gets updated with this value. When the strictness attribute appears without arguments, i.e. the `div` instruction, it means that all the operands reduce to *KResult* values, before further computation takes place.

2.2 Configuration

The configuration the *SSRISC* assembly language consists of a set of general integer and floating point registers, a number of special registers, a memory representation - usually as code and various data segments etc. Leaving aside the semantic entities for the memory modeling for reasons that we explain later, the *SSRISC* language configuration is the following:

SYNTAX *Instr* ::= `add` *Reg* `,` *Reg* `,` *Reg* `;` [strict(2 3)]
| `div` *Reg* `,` *Reg* `;` [strict(1 2)]
| `j` *Addr* `;` [strict]
| `jal` *Addr* `;` [strict]
| `beq` *Reg* `,` *Reg* `,` *Addr* `;` [strict(1 2)]
| `lw` *Reg* `,` *Off* `(` *Reg* `);` [strict(3)]
| `sw` *Reg* `,` *Off* `(` *Reg* `);` [strict(3)]
| `break;`
| `bc1t` *Off* `;`
| `l.s` *FReg* `,` *Off* `(` *Reg* `);` [strict(3)]
| `s.s` *FReg* `,` *Off* `(` *Reg* `);` [strict(3)]
| `add.s` *FReg* `,` *FReg* `,` *FReg* `;` [strict(2 3)]
| `div.s` *FReg* `,` *FReg* `,` *FReg* `;` [strict(2 3)]

Fig. 2. *SSRISC* abstract syntax: *Reg* and *FReg* means integer and respectively floating point registers, *Addr, Off* are of sort *#Int32*

CONFIGURATION:

$\langle \cdot \rangle_{\mathsf{k}}\ \langle 0 \rangle_{\mathsf{pc}}\ \langle 0 \rangle_{\mathsf{hi}}\ \langle 0 \rangle_{\mathsf{lo}}\ \langle 0 \rangle_{\mathsf{ra}}\ \langle 0 \rangle_{\mathsf{break}}\ \langle \cdot \rangle_{\mathsf{regs}}\ \langle 0 \rangle_{\mathsf{fcc}}\ \langle \cdot \rangle_{\mathsf{fregs}}$

$\mathbb{K}$ uses two special cells: `T` (not represented) encloses all the other cells and `k` maintains computational contents, such as programs or program fragments. `pc` is a special register, called program counter and its value indicates the current executing instruction. We opt to represent the program counter in a different cell than the other registers, as it improves the readability of the semantics, especially on conditional and unconditional jumps. `lo` and `hi` are special registers, required by the integer multiplication and division instructions to hold parts of the computed results. The `ra` cell represents a special register which stores the return address of a function call. The `fcc` is a special flag used in comparison operations between floating point registers. The `regs` and `fregs` cells consists of the set of the integer registers, respectively the floating point registers, as mappings from integer, respectively floating point, register names to stored values. The `break` cell is used only by the instruction `break` and captures program errors such as overflow or division by zero.

2.3 Semantics

The execution of each *SSRISC* instruction is split in a number of successive steps. It starts with the requests for the instruction and the operands from the memory, then continues with the actual processing and terminates with the state update. In this paper we adopt a sequential model of execution for the *SSRISC* programs, the integration of the pipeline-based computation model, supported by the MIPS languages is the subject of subsequent investigation.

The computational layer of the semantics contains some common functionality (or support operations), as general register lookup and update, or use some wrappers as for the `pc` register update. Before we describe the instruction semantics, we cover these general rules. For example, to execute the instruction at the current program point, denoted by the value of *PC* in the `pc` cell, the rule below the computation evolves to `geti(PC)` which is an instruction request from the memory.

RULE $\langle \underbrace{\cdot}_{\texttt{geti(}\ PC\ \texttt{)}} \rangle_{\texttt{k}}\ \langle PC \rangle_{\texttt{pc}}$

The rule reads like this: the underlined k cell emphasizes a "write" part of the term and information below the line, `geti(PC)` replaces the empty computation, represented with a dot. The pc cell represents a "read" part of the term, it is not underlined, and provides the value PC that is used by `geti`.

The lookup and update operations on the registers require two cells, k and regs for integer registers or k and fregs for the floating point registers. We have included only the rule for the integer registers case. If the current computational task is an integer register lookup, for a register R, as shown below the resulting configuration has the corresponding value I for R from the integer register file.

RULE $\langle \underbrace{R}_{I}\ \cdots \rangle_{\texttt{k}}\ \langle \cdots\ R \mapsto I\ \cdots \rangle_{\texttt{regs}}$

This rule brings a new element of the $\mathbb{K}$ notation, the "don't care" part of a term, represented with the dots. If this follows a subterm, such as register R in k cell, it means that R is the first computational task, followed by potentially other computational tasks, which are not important for this rule. Similarly, if the top computational task is to update a register, say Rd with a computed value I, the previous value of Rd, denoted by the wildcard _, is replaced by I, as shown below. The regs cell has dots at both ends and represents that the element $R \mapsto I$ is not necessarily the first nor the last in the cell.

RULE $\langle \underbrace{\texttt{updateReg(}\ I_1\ \texttt{,}\ Rd\ \texttt{)}}_{\cdot} \rangle_{\texttt{k}}\ \langle \cdots\ Rd \mapsto \underbrace{\_}_{I_1}\ \cdots \rangle_{\texttt{regs}}$

The pc register update consists of three cases: the rule shown below with the automated incrementation before an instruction fetch, the case of a mandatory jump which results into the pc cell information update and finally, the fall-through case which leaves the value of pc unmodified.

RULE $\langle \underbrace{\texttt{incPC(}\ PC\ \texttt{)}}_{\cdot}\ \cdots \rangle_{\texttt{k}}\ \langle \underbrace{PC}_{PC\ \texttt{+Int32 4}} \rangle_{\texttt{pc}}$

We present a number of *SSRISC* instructions, which cover arithmetic-logic instructions on both integer and floating point registers, branch and jump instructions, load and store instructions, an instruction to manipulate special flags (i.e. `bc1t`) as well as a special instruction for program errors - `break`. We implement the semantics in [4].

The $\mathbb{K}$ semantic rules having an instruction on top of the k cell transform it into either a register update or an error check followed by another register operation etc. All these rules are labeled as structural and form one layer of the semantics.

Out of the arithmetic-logic instructions, we present the two addition instructions: `add` on integer operands and `add.s` on floating point operands. The $\mathbb{K}$ rule for `add` states that the instruction with the source integer registers having values V_1 and V_2 reduces to an overflow check, `ovf`, for the signed addition between these two values and, if necessary, followed by the destination register Rd update with the result. When the overflow condition is on top of the computation and it is evaluated to true, the execution continues with an instruction for program errors, `break`. The $\mathbb{K}$ rule for addition instruction `add.s` is reduced to a term for the register update, `updateFReg`,

having the single precision sum of the two floating point values F_1 and F_2. The semantic definitions of these two instructions rely on support operations: `+Int32`, `==Int32` for the sum and comparison between 32-bit signed integers, `+SgFloat` for the sum of single precision floating point values.

RULE $\langle$ `add` Rd `,` V_1 `,` V_2 `;` $\rangle_{\mathsf{k}}$ [structural]
$\overset{}{\mathtt{ovf}(\ V_1\ ,\ V_2\ )} \curvearrowright \mathtt{updateReg}(\ V_1\ \mathtt{+Int32}\ V_2\ ,\ Rd\ )$

RULE $\langle \mathtt{ovf}(\ V_1\ ,\ V_2\ )\ \curvearrowright \rangle_{\mathsf{k}}$ when $\mathtt{ovfs32}\ (\ V_1\ ,\ V_2\ )\ \mathtt{==Int32}\ 1$
$\mathtt{break};$

RULE $\langle$ `add.s` Fd `,` F_1 `,` F_2 `;` $\rangle_{\mathsf{k}}$ [structural]
$\mathtt{updateFReg}(\ F_1\ \mathtt{+SgFloat}\ F_2\ ,\ Fd\ )$

The two division instructions, `div` and `div.s` are transformed into a division by zero check for the denominator value followed by the register updates, the special registers `lo` and `hi`, respectively a general purpose floating point register Fd.

The branch and jump instructions transform the task in the `k` cell into a correct `pc` register update. Before each instruction is reduced, via a structural $\mathbb{K}$ rule, into the corresponding operation, the value of the `pc` register has the address of the next instruction. The two rules for instructions `j` and `jal` use the `setPC` operation with the first argument 1 which means a program counter update. The jump and link `jal` instruction saves, in the `ra` cell, the return address after the execution of the callee function terminates. The instruction `bc1t` sets the target address depending on the value of a special flag called `fcc`.

RULE $\langle$ `j` $Addr$ `;` $\rangle_{\mathsf{k}}$ [structural]
$\mathtt{setPC}(\ 1\ ,\ Addr\ )$

RULE $\langle$ `jal` Imm `;` $\rangle_{\mathsf{k}}\ \langle PC \rangle_{\mathsf{pc}}\ \langle\ \_\ \rangle_{\mathsf{ra}}$ [structural]
$\mathtt{setPC}(\ 1\ ,\ Imm\ )\qquad PC\ \mathtt{+Int32}\ 8$

RULE $\langle$ `bc1t` Off `;` $\rangle_{\mathsf{k}}\ \langle FC \rangle_{\mathsf{fcc}}$ [structural]
$\mathtt{setPC}(\ \mathtt{Bool2Int}(\ FC\ \mathtt{==Bool}\ 1\ )\ ,\ Off\ )$

For the branch when equal `beq`, the address of the next instruction depends on the comparison between two values, V_1 and V_2, fall-through for 0 or branch taken for 1. `Bool2Int` is a built-in operation to ensure correct sort coercion.

The *SSRISC* assembly language has two families of load/store instructions: `lw` and `sw` use integer registers as data source, respectively data destination and `l.s` and `s.s` manipulate, in the same way, floating point registers. Both load instructions are reduced to memory read requests via `getd` operation, which takes two arguments, the memory address and the destination register (Rd or Ft). Similarly, the store instructions are reduced to memory write requests via `putd` operation, which has the memory address and the source register as arguments. The main memory, presented in the next section, processes the `getd` and `putd` requests.

RULE $\langle$ `l.s` Ft `,` Off `(` V_1 `);` $\rangle_{\mathsf{k}}$ [structural]
$\mathtt{updateFReg}(\ \mathtt{getd}(\ V_1\ \mathtt{+Int32}\ Off\ )\ ,\ Ft\ )$

RULE $\langle$ `s.s` Ft `,` Off `(` V_1 `);` $\rangle_{\mathsf{k}}$ [structural]
$\mathtt{putd}(\ V_1\ \mathtt{+Int32}\ Off\ ,\ Ft\ )$

The last semantic rule of the *SSRISC* language treats the special `break` instruction. The `k` cell gets the `last` term that ends the computation, while the special `break` cell updates to reflect a program error. We mention that `last` is also used for normal termination of computation.

RULE $\langle \overset{\curvearrowright}{\texttt{break;}}\ \cdots\rangle_{\texttt{k}}\ \langle\ \overset{\curvearrowright}{\underline{\ \ }}\ \rangle_{\texttt{break}}$ [structural]

The *SSRISC* language has also a number of pseudo-instructions, each being a syntactic sugar for a sequence of instructions. In the next section we introduce and discuss, in the context of main memory modeling, one important pseudo-instruction called `la` - load address.

3 The $\mathbb{K}$ Memory Modeling

Our design relies on a number of modules which communicate using predefined message names. We recall that the concrete configuration, described in Section 2.3, omits a store or memory cell which is actually necessary to capture program executions. We decide to design the language semantics rules to update only the registers and, in this way, to decouple the representation of the main memory. This makes possible to represent, in an independent way, both the labeled representation of data and instruction addresses as well as the memory layout of the disassembled code and data. We start with the former.

3.1 Memory Model

We emulate the organization of an assembly file into a code and data text, with the $\mathbb{K}$ configuration for the main memory having, along with the k cell, the two corresponding cells, cmem and respectively dmem.

CONFIGURATION:

$\langle \cdot \rangle_{\texttt{k}}\ \langle \cdot \rangle_{\texttt{cmem}}\ \langle \cdot \rangle_{\texttt{dmem}}$

The k cell processes the requests for instruction or data that come from the cache memories or the processor. In our design, the language semantics issues an instruction request, using the `geti`(PC) operation. The memory system interprets the PC value as an address and checks this location in the code memory part, cmem cell. There are two possible cases, each modeled with a $\mathbb{K}$ rule. If the instruction is found in the code memory cmem, and `geti`(PC) rewrites to the actual instruction, while the control is back to the processor. If the instruction is not found in the code memory cmem, a special token denoted as `last`, signals the execution termination. We rely on a special built-in function `notIn` to check if the instruction exists in the code memory.

RULE $\langle \overset{\curvearrowright}{\texttt{incPC(}\ PC\ \texttt{)} \curvearrowright Ins}\ \cdots\rangle_{\texttt{k}}\ \langle \cdots\ PC \mapsto Ins\ \cdots\rangle_{\texttt{cmem}}$

RULE $\langle \overset{\curvearrowright}{\texttt{geti(}\ PC\ \texttt{)}}\rangle_{\texttt{k}}\ \langle CMem\rangle_{\texttt{cmem}}$ when $\texttt{notIn}\ (\ CMem\ ,\ PC\ )$

We handle memory data requests using the memory address $Addr$ wrapped using a special communication message `getd`. There are two cases. The first and obvious one is when the data memory, in the cell dmem has the necessary data, $Data$. The second case assumes that, the data was not previously initialized or written, a new memory address Addr is added in the dmem cell and the value is 0.

RULE $\langle \overset{\curvearrowright}{\texttt{getd(}\ Addr\ \texttt{)}}\ \cdots\rangle_{\texttt{k}}\ \langle \cdots\ Addr \mapsto Data\ \cdots\rangle_{\texttt{dmem}}$

SYNTAX $Instr ::=$ `la` Reg `,` $SVar$ `;`
| `lw` Reg `,` $SVar$ `+` Off `;`

Fig. 3. *Extension of SSRISC abstract syntax: the sort* `SVar` *represents for symbolic data address*

RULE $\langle \underset{\check{0}}{\texttt{getd(}\ Addr\ \texttt{)}} \cdots \rangle_{\texttt{k}}\ \langle DMem\ \underset{Addr \mapsto 0}{\cdot} \rangle_{\texttt{dmem}}$ when $\texttt{notIn(}\ DMem\ \texttt{,}\ Addr\ \texttt{)}$

An alternative is to initialize such a data with an unknown, symbolic value, which can be accommodated with changes at the built-in level. We refer to this issue in the implementation section.

3.2 An Extended Memory Model

We introduce next the labeled representation of a memory addresses and how the $\mathbb{K}$ framework caters for easy semantic extensions, both at the level of configuration and rules. In Fig. 1 (middle) there is an assembly language snippet that uses labels for instruction addresses (i.e. L_1, L_2) and for data addresses, (i.e. a).

For presentation purposes, we describe how to incrementally extend the *SSRISC* assembly language definition to accommodate labels for data addresses, assuming that all the instruction addresses are a priori assigned to actual memory addresses. Therefore, the two previous rules with the `geti` operation in the top of the `k` cell remain unmodified. The data request rules get extended with a special case. We describe next how our design can accommodate changes, from syntax augmentation, in Fig. 3, to modifications in the language semantics and in the memory model.

First, the new instruction `la` that loads a destination integer register with the data that resides at a symbolic address. The sort of this address is denoted by `SVar` and is subsorted to the built-in sort for identifiers `#Id`. Then, the load instruction `lw` could handle symbolic addresses, based on a specified offset.

The execution of an instruction that manipulates symbolic data addresses (i.e. `la` or `lw`) is transformed into an integer register Rd update, via a structural $\mathbb{K}$ rule. The symbolic memory address VA is wrapped by a new term, `getAddr`, which acts as communication message between the semantic and the memory model.

RULE $\langle \underset{\texttt{updateReg(}\ \widehat{\texttt{getAddr(}\ VA\ \texttt{)}}\ \texttt{,}\ Rd\ \texttt{)}}{\texttt{la}\ Rd\ \texttt{,}\ VA\ \texttt{;}} \rangle_{\texttt{k}}$ [structural]

RULE $\langle \underset{\texttt{updateReg(}\ \widehat{\texttt{getAddr(}\ VA\ \texttt{+}\ Off\ \texttt{)}}\ \texttt{,}\ Rd\ \texttt{)}}{\texttt{lw}\ Rd\ \texttt{,}\ VA\ \texttt{+}\ Off\ \texttt{;}} \rangle_{\texttt{k}}$ [structural]

We extend the memory model with a new cell, called `dlabels` which holds a mapping between symbolic addresses and concrete values. These values are arbitrarily generated and used in the `dmem` cell.

CONFIGURATION:

$\langle \cdot \rangle_{\texttt{k}}\ \langle \cdot \rangle_{\texttt{cmem}}\ \langle \cdot \rangle_{\texttt{dmem}}\ \langle \cdot \rangle_{\texttt{dlabels}}$

There are several other possible extensions. For example, the labels for instruction addresses can be handled in a similar way, using a specialized cell. The message `getAddr` is transformed into a lookup in the list of labels, to identify the value Val associated with a particular label VA. When this particular message is the result of the execution of a `lw` instruction, the actual data address is translated, using the information stored in the cell `dlabels`.

RULE $\langle \underline{\texttt{getAddr(}\ VA\ \texttt{)}}_{Val}\ \cdots\rangle_{\texttt{k}}\ \langle\cdots\ VA \mapsto Val\ \cdots\rangle_{\texttt{dlabels}}$

RULE $\langle \underline{\texttt{getAddr(}\ VA\ \texttt{+}\ V_1\ \texttt{)}}_{Val}\ \cdots\rangle_{\texttt{k}}\ \langle\cdots\ VA \mapsto V_2\ \cdots\rangle_{\texttt{dlabels}}\ \langle\cdots\ V_1\ \texttt{+Int32}\ V_2 \mapsto Val\ \cdots\rangle_{\texttt{dmem}}$

The set of instructions that appear in the labeled representation of the assembly code include, among others, the `lw` and `sw` instructions which manipulate the actual data addresses. Therefore, the extended memory module includes the $\mathbb{K}$ rules for the aforementioned communication messages `geti` and `getd`.

4 Implementation

The current implementation of the *SSRISC* assembly language follows a standard approach for programming language definitions in the $\mathbb{K}$ framework. In general, such a definition in $\mathbb{K}$ comprises of a syntax module, a semantic module and the use of a built-in module for the implementation of the support operations. Our definition of the *SSRISC* assembly language poses two kinds of particularities, imposed on one hand by the language specifics and on another hand by the target applications. We start with the former and elaborate on the later afterwards.

Our system design consists of several important modules, shown in Fig. 4. There are: a built-in module *builtins* and its extension to symbolic values and operations *s-builtins*, a "glue" module, *ssrisc-settings* to specify how the language semantics, represented by *ssrisc-lang* communicates with the main memory, represented by *ssrisc-mem*.

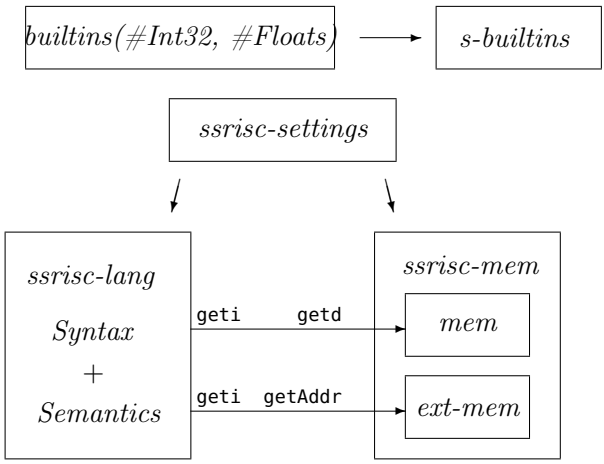

Fig. 4. The system organization: *ssrisc-settings* keeps the communication channels between the language definition *ssrisc-lang* and various main memory models *mem* and *ext-mem*

The language specifics influence the language design in the following way: the semantic rules of the assembly language instructions use various representations for integers (signed and unsigned, on 32-bit, 16-bit or 8-bit format) and floating point numbers (single and double precisions). All these, together with constant declarations, arithmetic and logic operations, conversion operations between these

data types and auxiliary checks (i.e. overflow/underflow, comparison with zero) are in a special built-in module, developed on top of the standard programming language built-in provided by $\mathbb{K}$. For example, the signed addition between two integers, `+Int32` is implemented over the `+Int` operation. The current implementation uses the predefined Maude number representations for both integers and floating points.

Apart from the implementation of a specific built-in module, we decide to separate the memory component from the other semantic pieces that define the *SSRISC* language state. We opt for this modular implementation to accommodate further refinements of the memory, with minimal modification of the language semantics.

In the previous section we presented such a refinement, that allows the representation of the data memory addresses with labels. We use this to construct the initial state of a program and to facilitate testing the *SSRISC* semantics. The alternative memory definition, which we obtain from disassembling the executables, does not help with this aspect. For the example program in Fig. 1 (middle), `dlabels` contains the information $a \mapsto 100$, and `dmem` contains $\{100 \mapsto -2, 104 \mapsto 1, 108 \mapsto 4\}$, in other words, the values of the array a - same figure (left). The symbolic address is transformed into concrete value, for example 100.

An important element in our design is what we name a communication module, to allow easy extensions to the definition. For example, the $\mathbb{K}$ rules for the load and store instructions use `geti` and `getd` operations to fetch instructions, respectively data, from the memory. We call these communication channels between the semantics and the memory module, and we place them into the communication module. In this way, the memory model could be replaced with a different one, without modifying the semantics. Another example is with respect to the labeled representation of the data memory addresses. The instruction `la` uses a communication channel called `getAddr` to access the information in the cell `dlabels`.

A particularity of the *SSRISC* definition is that it allows to underspecify the memory content using symbolic values for stored data. For example, if a load instruction (i.e. `lw`, `l.s` and `la` accesses a memory address with an unknown value, this value is retrieved and further propagated during the program execution. The set of specific built-in operations on bitwise representation is extended to handle the symbolic value. We have used this feature in the context of abstract execution of programs for estimation of timing bounds [1].

The Simplescalar toolset [4] is an architecture simulator and presents two sets of instructions: a MIPS-based assembly language, used to compile C programs into it and a set of simulator-specific instructions. We implement the former, together with a number of pseudo-instructions and some of the instructions specific to the program with labels. There are 112 instructions, each implemented using exactly one $\mathbb{K}$ rewrite rule, and 20 rewrite rules for auxiliary operations (i.e. set the program counter, overflow check etc). The memory modelings comprises of 15-20 rewrite rules, which are split into memory read and write cases (for word and double word).

Besides of the possibility to manually construct the initial state (i.e. registers and data memory contents) of the program, we allow a limited amount of automated testing, subjective to certain assumptions. For example, we need to produce assembly

code without library function calls (i.e. memcpy), which currently, we do not support. We consider simple C programs, with small arrays of integer and floating point values, to test various arithmetic and logic instructions, as well as load/store instructions and conditional and unconditional jumps. Also, the return value of a C program and its corresponding representation in the assembly program - the value in a particular register - should be compared and decide if the test program passes or fails. We reiterate that our intention is to present the *SSRISC* language definition as a basis to define abstractions for timing analysis of embedded programs. For this purpose, the input program uses the non-labeled representation of the main memory because it contains useful information about actual instruction and data placement.

5 Conclusions

Program reasoning at the assembly language level is particularly important in the security and embedded systems research areas. We approach this problem from a novel perspective, advocated by the $\mathbb{K}$ framework, to define programming languages and to use this definition for development of program analysis and verification tools. In this paper, we proposed a formal executable semantics of a RISC assembly language, called *SSRISC* and based on MIPS IV. We relied on the modularity of $\mathbb{K}$ to give two memory models, one using labels for data addresses and another for the addresses in the disassembled code. Also, the semantic definition required a specific built-in module for operations on the signed and unsigned integers and floating point numbers on single and double precisions. Everything was integrated and experimented with, in the $\mathbb{K}$-Maude tool.

In the strict sense of the definition of *SSRISC*, this current work should be extended in two directions. First, the implementation of the built-in module uses Maude's representation of integer and floating point numbers and therefore, it requires concrete implementations of some of the operations. There are two possible solutions to this: a native Maude implementation or using a third party program using the IO capabilities of $\mathbb{K}$ or Maude. Second, the main memory modeling is more complex than what we presented. For example, the actual MIPS data memory has a data segment where the program data is, a heap for extra space and a stack to handle subroutine calls. In our modeling we do not formally distinguish between these. Our desiderate is to use the formal executable semantics of *SSRISC* for the design and analysis of non-functional requirements (i.e. time, energy) of the embedded software systems.

Acknowledgement

The author would like to thank the anonymous reviewers for the valuable comments and suggestions to improve this work.

References

[1] Mihail Asavoae and Dorel Lucanu and Grigore Roşu, *Towards Semantics-Based WCET Analysis*, WCET, (2011)

[2] Todd M. Austin and Eric Larson and Dan Ernst, *SimpleScalar: An Infrastructure for Computer System Modeling*, IEEE Computer **35**, (2002), 59–67

[3] Bishop Brock and Warren A. Hunt Jr. *Formally Specifying and Mechanically Verifying Programs for the Motorola Complex Arithmetic Processor DSP*, ICCD, (1997), 31-36

[4] Doug Burger and Todd M. Austin, *The SimpleScalar tool set, version 2.0*, SIGARCH Comput. Archit. News **25**, (1997), 13–25

[5] Mihai Christodorescu and Somesh Jha and Christopher Kruegel *Mining specifications of malicious behavior*, ESEC-FSE, (2007), 5-14

[6] Manuel Clavel and Francisco Durán and Steven Eker and Patrick Lincoln and Narciso Martí-Oliet and José Meseguer and Carolyn L. Talcott, *All About Maude - A High-Performance Logical Framework, How to Specify, Program and Verify Systems in Rewriting Logic*, Springer LNCS **4350**, (2007)

[7] Manuel Clavel and Francisco Durán and Steven Eker and Patrick Lincoln and Narciso Martí-Oliet and José Meseguer and Jose F. Quesada, *The Maude System*, RTA, (1999), 240–243

[8] Chucky Ellison and Grigore Roşu, *An Executable Formal Semantics of C with Applications*, POPL, (2012), to appear

[9] Anthony C. J. Fox, *Formal Specification and Verification of ARM6*, TPHOL, (2003), 25-40

[10] Anthony C. J. Fox and Magnus O. Myreen, *A Trustworthy Monadic Formalization of the ARMv7 Instruction Set Architecture*, ITP, (2010), 243-258

[11] Michael J. C. Gordon and Thomas F. Melham, *Introduction to HOL: a Theorem Proving Environment for Higher Order Logic*, Cambridge University Press, (1993)

[12] Neal A. Harman, *Verifying a Simple Pipelined Microprocessor Using Maude*, WADT, (2001), 128–151

[13] Mark Hills, *Memory Representations in Rewriting Logic Semantics Definitions*, WRLA, (2008), 155-172

[14] C.A.R. Hoare, *The Verifying Compiler, a Grand Challenge for Computing Research*, VMCAI, (2005), 78–78

[15] Matt Kaufmann and J. Strother Moore and Panagiotis Manolios, *Computer-Aided Reasoning: An Approach*, Kluwer Academic Publishers, (2000)

[16] Xavier Leroy, *Formal verification of a realistic compiler*, Commun. ACM, **52**, (2009), 107-115

[17] Cullen Linn and Saumya Debray, *Obfuscation of executable code to improve resistance to static disassembly*, Conference on Computer and Communications Security, (2003), 290-299

[18] Narciso Martí-Oliet and José Meseguer, *Rewriting logic: roadmap and bibliography*, Theor. Comput. Sci. , **285**, (2002), 121-154

[19] Patrick Meredith and Mark Hills and Grigore Roşu, *An Executable Rewriting Logic Semantics of K-Scheme*, SCHEME'07, (2007), 91–103

[20] Patrick Meredith and Michael Katelman and José Meseguer and Grigore Roşu, *A Formal Executable Semantics of Verilog*, MEMOCODE, (2010), 179–188

[21] José Meseguer and Grigore Roşu, *The Rewriting Logic Semantics Project*, Electronic Notes in Theoretical Computer Science **156** (2006), 27–56.

[22] Greg Morrisett and Karl Crary and Neal Glew and Dan Grossman and Richard Samuels and Frederick Smith and David Walker and Stephanie Weirich and Steve Zdancewic, *TALx86: A Realistic Typed Assembly Language*, In Second Workshop on Compiler Support for System Software, (1999), 25–35

[23] Grigore Roşu and Traian Florin Şerbănuţă, *An Overview of the K Semantic Framework*, Journal of Logic and Algebraic Programming **79**, (2010), 397–434

[24] Roşu, Grigore and Ellison, Chucky and Schulte, Wolfram, *Matching Logic: An Alternative to Hoare/Floyd Logic*, AMAST, LNCS **6486**, (2011), 142–162

[25] Traian Şerbănuţă and Andrei Arusoaie and David Lazăr and Chucky Ellison and Dorel Lucanu and Grigore Roşu *The K primer (version 2.5),* Submitted.

[26] Traian Şerbănuţă and Grigore Roşu and Andrei Ştefănescu *An Overview of K and Matching Logic,* Submitted.

[27] Matt Webster and Grant Malcolm, *Detection of metamorphic and virtualization-based malware using algebraic specification*, Journal in Computer Virology **5**, (2009), 221–245

[28] Wilhelm, Reinhard and Engblom, Jakob and Ermedahl, Andreas and Holsti, Niklas and Thesing, Stephan and Whalley, David and Bernat, Guillem and Ferdinand, Christian and Heckmann, Reinhold and Mitra, Tulika and Mueller, Frank and Puaut, Isabelle and Puschner, Peter and Staschulat, Jan and Stenström, Per, *The worst-case execution-time problem—overview of methods and survey of tools*, ACM Trans. Embed. Comput. Syst. **7**, (2008), 1–53