Institute for System Programming of the Russian
Academy of Sciences

nML Language Reference
(UNDER DEVELOPMENT)

Moscow 2016

Contents

Chapter 1

Introduction

1.1 Disclaimer

This reference describes syntax and semantics of the nML architecture description language. It covers only language facilities supported by the ISP RAS version of nML. It may differ from documentation provided by other vendors as there are differences in language implementations.

1.2 Overview

nML is an architecture description language (ADL) used to describe the instruction set architecture (ISA) of an arbitrary microprocessor. It is a flexible and easy-to-use language based on attribute grammar. nML was designed to provide a retargetable way to specify microprocessor architecture for various microprocessor-related software tools including instruction-set simulators, assemblers, disassemblers, compiler back-ends etc. It works at the instruction-set level concentrating on behavioral properties and hiding implementation details. An nML specification represents a programmer's model of the microprocessor which covers the following aspects:

- supported data types;
- registers and memory;
- addressing modes;
- syntax and semantics of instructions.

nML uses a hierarchical tree-like structure to describe an instruction set. Such a structure facilitates grouping related instructions and sharing their common parts. An instruction is described as a path in the tree from the root node to a leaf node. The set of all possible paths represents an instruction set. A node describes a primitive operation responsible for some task within an instruction. Each node has certain attributes that can be shared with its descendants. Actions performed by instructions are described as operations with registers and memory represented by bit vectors of arbitrary size.

A specification in nML starts with definitions of types and global constants. For example, a type definition for a 32-bit unsigned integer looks as follows:

```
let WORD_SIZE = 32
type WORD = card(WORD_SIZE)
```

Type definitions and constants can be used to describe registers and memory. In addition to registers and memory, it is also possible to define temporary variables that are internal abstractions provided by nML to store intermediate results of operations. They do not correspond to any data storage in real hardware and do not save their data across instruction calls. Also, there is often a need to specify some properties of the described model. For this purpose, special constants are used. For example, the code below defines general-purpose registers, memory and a temporary variable. Also, it includes a special constant to establish a correspondence between the general purpose register number 15 and the program counter (PC). Here is the code:

```
reg GPR[32, WORD]
mem M[2 ** 20, BYTE]
var carry[1, BIT]
let PC = "GPR[15]"
```

As stated above, an instruction set is described as a tree of primitive operations. There two kinds of primitives: operations and addressing modes. Operations describe parts of instructions responsible for specific tasks and can be used as leaf and root nodes. Addressing modes are aimed to customize operations (for example, they encapsulate rules for accessing microprocessor resources). They can only be used as leaf nodes. For example, here are simplified examples of operation and addressing mode specifications:

```
mode REG(i: nibble) = R[i]
syntax = format("R%d", i)
image  = format("01%4b", i)
```

```
op Add()
syntax = "add"
image  = "00"
action = { DEST = SRC1 + SRC2; }
```

Operations and addressing modes have three standard attributes: syntax, image and action. The first two specify textual and binary syntax. The third describes semantics of the primitive. In addition, addressing modes have a return expression that enables them to be used as variables in various expressions. Attributes can be used by parent primitives referring to a given primitive to describe more complex abstractions.

Primitives are arranged into a tree using production rules. There are two kinds of production rules: AND rules and OR rules. AND rules specify parent-child relationships where a child primitive is described as a parameter of its parent. Here is an example of an AND rule:

```
op arith_inst(act: Add, op1: OPRND, op2: OPRND)
```

This is the header of the `arith_inst` operation that states that the `arith_inst` operation node has three child nodes: the `act` operation and the `op1` and `op2` addressing modes. The syntax of an operation header is similar to a function where parameter types specify the primitives the rule refers to. Parameter can be, in turn, parameterized with other primitives (they will be encapsulated behind attributes). For this reason child nodes represent independent instances that are accessed from their parent node via parameters. OR rules specify alternatives. This means that a group of primitives is united under some alias so that each of them can used when this alias is specified in an AND rule. An OR rule looks as follows:

```
op Add_Sub_Mov = Add | Sub | Mov
```

Figure 1 displays a tree path describing the `mov` instruction from an imaginary instruction set. This instruction copies data from one register to another. The root operation of the instruction is called `instruction`. According to nML conventions, the root operation is always called `instruction` and it specifies the point from which all paths describing specific instructions start. The `instruction` operation can be defined either as AND or OR rule. In the latter case, there are several starting points. Usually root operations holds some common properties and perform common actions (such as increment of the program counter). In the given example, the root operation is linked to

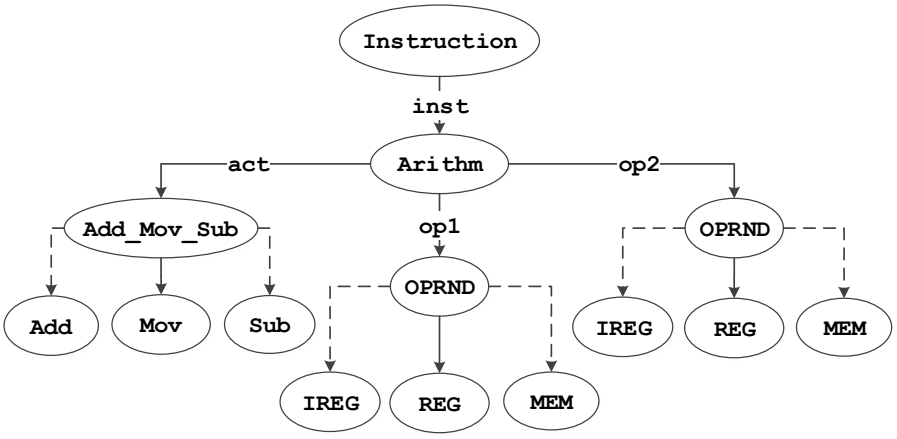

Figure 1.1: Operation tree for the Mov instruction

the `Arithm` operation with the help of an AND rule. This operation describes a group of arithmetic operations. It is parameterized with the `Add_Mov_Sub` and `OPRND` primitives. Both of them are specified as OR rules. The first one describes arithmetic operations that can be performed by the `Arithm` primitive while the second one specifies supported addressing modes. Dashed lines that connect OR-rules with their child primitives specify possible alternative paths. Instructions are identified by the terminal operation node of the path (in this example, it is the `Mov` node). An important note is that, to avoid ambiguity, nodes can have only one child operation.

The syntax of nML resembles the syntax of the pseudocode used in microprocessor architecture manuals to describe instruction semantics. For example, here is the description of instruction ADD from the MIPS64 manual:

```
if NotWordValue(GPR[rs]) or NotWordValue(GPR[rt]) then
  UNPREDICTABLE
endif
temp ← GPR[rs]31||GPR[rs]31..0) + (GPR[rt]31||GPR[rt]31..0)
if temp32 ≠ temp31 then
  SignalException(IntegerOverflow)
else
  GPR[rd] ← sign_extend(temp31..0)
endif
```

Such a description can be translated to nML with minimal effort. Providing that all needed data types, resources and operations describing common functionality of instructions have already been specified, the specification of the ADD instruction (or, to be more precise, the terminal operation that distinguishes it from other similar instructions) will look

as follows:

```
op add (rd: R, rs: R, rt: R)
  syntax = format("add %s, %s, %s", rd.syntax, rs.syntax, rt.syntax)
  image  = format("000000%5s%5s%5s00000100000", rs.image, rt.image, rd.image)
  action = {
    if sign_extend(WORD, rs<31>) != rs<63..32> ||
       sign_extend(WORD, rt<31>) != rt<63..32> then
      unpredicted;
    endif;

    temp33 = rs<31>::rs<31..0> + rt<31>::rt<31..0>;
    if temp33<32> != temp33<31> then
      C0_EPC = CIA;
      exception("IntegerOverflow");
    else
      rd = sign_extend(DWORD, temp33<31..0>);
    endif;
  }
```

As we can see, describing an instruction based on an instruction set manual is a relatively easy task that can be performed by a verification engineer who does not have significant programming skills.

Chapter 2

Language Facilities

2.1 Constants

In nML, constants are declared using the ***let*** keyword. For example, the following line of code declares a constant called `A` that has the value `100`:

```
let A = 100
```

Constants are global. They can be defined only once and can be used in any context their values could stand.

Constants can help extend nML. Any information about a machine that can be given with a single number or string can easily be defined as a constant.

2.1.1 Constant Types

Constants can be represented by integer numbers of inifinite precision and strings. Integer constants are cast to specific nML data types depending on the context in which they are used.

2.2 Data Types

A data type specifies the format of values stored in registers or memory. nML supports the following data types:

- ***int(N)***: N-bit signed integer data type. Negative numbers are stored in two's complement form. The range of possible values is [-2^{n-1} ... 2^{n-1} - 1].

- ***card(N)***: N-bit unsigned integer data type. The range of possible values is [0 ... 2^n - 1].
- ***float(N, M)***: IEEE 754 floating point number, where fraction size is N and exponent size is M. The resulting type size is N + M +1 bits, where 1 is an implicitly added bit for store the sign. Supported floating-point formats include:
  - 32-bit single-precision. Defined as float(23, 8).
  - 64-bit double-precision. Defined as float(52, 11).
  - 80-bit double-extended-precision. Defined as float(64, 15).
  - 128-bit quadruple-precision. Defined as float(112, 15).

nML allows declaring aliases for data types using the ***type*** keyword. For example, in the code below, WORD is declared as an alias for card(32):

```
type WORD = card(32)
```

2.2.1 Converting Data Types

In nML, type conversion is performed explicitly using special functions. Implicit type conversion is not supported. Type conversion functions:

- ***sign_extend(type, value)***: Sign-extends the value to the specified type. Applied to any types. Requires the new type to be larger or equal to the original type.
- ***zero_extend(type, value)***: Zero-extends the value to the specified type. Applied to any types. Requires the new type to be larger or equal to the original type.
- ***coerce(type, value)***: Converts an integer value to another integer type. For smaller types the value is truncated, for larger types it is extended. If the original type is a signed integer, sign extension is performed. Otherwise, zero-extension is performed.
- ***cast(type, value)***: Reinterprets the value as the specified type. Applied to any types. The new type must be of the same size as the original type.

- ***int_to_float(type, value)***: Converts 32 and 64-bit integers into 32, 64, 80 and 128-bit floating-point values (IEEE 754).
- ***float_to_int(type, value)***: Converts 32, 64, 80 and 128-bit floating-point values (IEEE 754) into 32 and 64-bit integers.
- ***float_to_float(type, value)***: Converts 32, 64, 80 and 128-bit floating-point values (IEEE 754) into each other.

2.3 Memory, Registers and Variables

Memory and registers represent the visible state of a microprocessor. Only this state is carried across execution of instruction calls. Both entities are modeled as arrays of the specified data type. Although memory is external to the microprocessor while registers are internal, they are described in nML in the same way. Differences are related to intergration with the MMU model and are transparent to users. In addition to memory and registers, nML supports defining temporary variables, which can be used to exchange data between primitives that constitute a microprocessor instruction. Such variables do not represent any externally visible state and do not save their values across instruction calls.

2.3.1 Memory

Memory is described as a data array that represents the state of physical memory. Virtual memory issues such as address translation, caching, etc. are not covered by nML specifications.

A memory array is defined using the ***mem*** keyword. For example, the code below defines a memory array called `MEM` that consists of 2^{30} words:

```
mem MEM [2 ** 30, WORD]
```

The array lenght must be specified as a contant expression. In the initial state, the array is filled with zeros.

2.3.2 Registers

Registers are defined using the ***reg*** keyword. Like memory, they are described as an array of the specified type. For example, the following

code defines a register array that includes 32 registers storing a word:

```
mem GPR[32, WORD]
```

The array length must be specified as a constant expression. Uninitialized registers hold zeros. When the array contains only a single element, the length expression can be skipped:

```
reg HI[WORD]
```

2.3.3 Variables

Variables are defined using the ***var*** keyword. Their role is to store temporary values used by instructions. They are similar to registers, but they do not represent any visible state and do not save their values across instruction calls. Variables are initialized with zeros and reset their values after the execution of an instruction call. For example, the code below defines a variable for storing a 33-bit signed integer value:

```
var temp[int(33)]
```

Chapter 3

Appendixes

3.1 Grammar of nML

3.2 References

Bibliography

[1] M. Freericks. *The nML Machine Description Formalism*. Technical Report TR SM-IMP/DIST/08, TU Berlin CS Department, 1993.