The `nML` Machine Description Formalism

TU Berlin Computer Science Technical Report — Updated & Revised Version 1.5

DRAFT

Markus Freericks

29.4.91 - 1.7.93

Abstract

`nML` is a formalism targetted for describing arbitrary single-processor computer architectures. `nML` works at the instruction set level, i.e. it hides implementation issues of the actual machine. `nML` can be used as an input language for a wide range of tools that need formal machine descriptions. Based on attribute grammars, `nML` is flexible and reasonably easy to use.

Contents

Preface

The nML machine description language has been changing since its inception in December, 1990. This report is a direct descendant of the first nML definition, which was printed as [1]. The 1.2 version of the report was included in the SPRITE 2260 progress report [2]. The changes between the different versions are recorded in section 8.

1 Introduction

1.1 Where are Machine Descriptions needed?

There are a lot of different software applications where detailed formal descriptions of computer architectures are needed. A few of these are:

- Simulators:
  in the development phase of an architecture, i.e. before actual hardware exists, instruction-level simulations are needed for writing the first actual programs and for testing compiler code generation.
- Assemblers and Dissassemblers:
  these programs are so simple that they ought to be easily generated automatically once a formal description of the assembler syntax and binary coding of the instruction set exist.
- Compiler back ends:
  modern compiler technology uses pattern-matched code-generation schemes. These patterns are usually written and optimized by hand and serve as "indirect" machine descriptions; the rest of the knowledge about the machine (e.g. pipeline behaviour and register allocation schemes) is hand-coded into the pattern-matcher or into special allocation and optimization passes. It should be possible to generate code-generation pattern libraries out of "functional" machine descriptions, i.e. one that are not centered on the special needs of the code generator. A more ambitious goal would be the generation of the whole code generator.

1.2 Different Kinds of Machine Descriptions

Many different kinds of machine descriptions are employed today. Two of the better known ones include:

1.2.1 GCC's `.md` format

GCC, the GNU C compiler, can be adapted to different machines by changing its machine description[1]. To quote from [13]:

> GNU CC gets most of the information about the target machine from a machine description which gives an algebraic formula for each of the machine's instructions. This

[1]At least when the machines don't derive too much from the built-in assumptions

is a very clean way to describe the target. But when the compiler needs information that is difficult to express in this fashion, I have not hesitated to define an ad-hoc parameter to the machine description.

[...]

A machine description has two parts: a file of instruction patterns ('.md' file) and a C header file of macro definitions.

The '.md' file for a target machine contains a pattern for each instruction that the target machine supports (or at least each instruction that is worth telling the compiler about).

[...]

Each instruction pattern contains an incomplete RTL expression, with pieces to be filled in later, operand constraints that restrict how the pieces can be filled in, and an output pattern or C code to generate the assembler output, all wrapped up in a '`define_insn`' expression.

[...]

Here is an actual example of an instruction pattern, for the 68000/68020.

```
(define_insn "tstsi"
  [(set (cc0)
        (match_operand:SI 0 "general_operand" "rm"))]
  ""
  "*
 {if (TARGET_68020 || ! ADDRESS_REG_P (operands[0]))
    return \"tstl %0\";
  return \"cmpl #0,%0\"; }")
```

RTL is the intermediate language of the compiler. A special program transforms a machine description into a C function that is used within the compiler. As can be seen in the example, the description, while being powerful – after all, any C function may be incorporated into the pattern-matching and expanding process –, is quite dependent on compiler internals and not very intuitive.

1.2.2 The VHDL Hardware Definition Language

VHDL is a language used for describing all kinds of digital circuits, among them processors and their components. A VHDL 'program' describes a circuit as a box having a number of input and output ports either by assembling it from other, previously defined boxes, or by giving 'the program that executes inside' the box. When using VHDL to describe a computer architecture, usually the whole data-path-, fetch-, decode-, load-, execute- and store-mechanism is described. In the simplest case, the processor is seen as a black box containing a large 'switch'-statement, i.e. the description really is a simulation program.

There are a number of other languages similar to VHDL, most notably the "Electronic Design Interchange Format" EDIF, which mainly describes the graphical layout of a circuit and has no 'semantic level' aside from the information about connections between predefined cells, and HILARICS-2, which is mostly equivalent to a human-readable version of the 'net'-part of EDIF.

VHDL is described in [5], EDIF in [8, 9] and HILARICS-2 in [12]. The authors of [4] specify a hypothetical processor using VHDL. They do this by describing everything down to the timing of the bus signals.

1.3 General aims of `nML`

In the following the main goals in the development of `nML` are presented together with the way of their realization.

1.3.1 Abstraction Level

The abstraction level `nML` aims at is that of the instruction set, i.e. the "programmer's model" of the processor. To program a machine, one needs to know about

- the memory model,
- different kinds of registers,
- directly supported data types,
- the exact semantics of instructions (including "side-effects"),
- addressing modes,
- alignment restrictions,
- condition code usage and
- processor-internal data structures like pipes.

One doesn't necessarily have to know about "system" programming details like exceptions and interrupts. A machine description should be as precise as necessary, but not more. E.g., in writing a machine description for an assembler/disassembler, the semantics of instructions can be ignored at all.

`nML` is based on a minimal set of assumptions about the machine: a machine, when run, executes a *program* that is a series of *instructions*. A *program counter* (*PC*) points to the next-to-be-executed instruction while executing the current one. A machine has *state* stored in *memory locations*. The sole purpose of a program is to change the contents of these locations.

All that instructions do is changing the values of locations. There are no inter-instruction control flow constructs; the program flow is changed by writing to the PC location. Each instruction can be seen as a function from state to state. By composing the instructions, the semantics of the whole program can thusly be given.[2] For practical reasons, instruction semantics are not given as one function, but as a sequence of assignments of the basic form

$$\textit{location} = \ \textit{function}\ (\ \textit{location}\ \ldots\ )$$

A finite set of primitive functions (arithmetic, shifting, masking) is assumed.

Traditionally, such assignment sequences are known as "register transfers". One early register transfer formalism was *ISP*. Variations of this formalism were used in retargetable code generation [6] and peephole optimization [7] systems.

[2]It is not that simple because of the program counter. While a semantic function of type *State* $\rightarrow$ *State* can be given easily for any program, it is of no great value, because this function has to be applied iteratively until some halting condition is reached.

1.3.2 Sharing in Descriptions

In complex architectures, there may be hundreds of different combinations of operations and addressing modes. If instructions or addressing modes have side effects (e.g., setting condition codes), the semantic description of a single complete instruction may grow quite large. One goal of in the development of `nML` was therefore the reduction of description size by sharing as much of descriptions as possible.

To achieve this goal, the instruction set is enumerated by an attributed grammar. E.g, a machine may have a dozen numeric instruction that share the behaviour of conditionally setting a 'zero flag'. This can be modelled by a grammar fragment

```
mem tmp_src[1,long]             \ temporary registers
mem tmp_dst[1,long]

\ grammar rule for binary numeric operations
\ SRC and DST are addressing modes, described elsewhere

op numeric_instruction(a:num_action,src:SRC,dst:DST)
action={
        tmp_src=src;
        tmp_dst=dst;
        a.action;              \ execute the numeric_action
        if tmp_dst==0          \ is the result zero?
         then CZ=1;            \ yes: set zero flag
         else CZ=0;            \ no: clear it
        endif;
        dst=tmp_dst;
       }

op num_action= add | sub | ...

op add()
action={
        tmp_dst=tmp_dst+tmp_src;
       }
```

The semantic action of any instruction is composed of fragments that are distributed over the whole grammar tree. This has been compared to the "inheritance" of object-oriented languages; in the above example, `numeric_instruction` is an "abstract base class" for `add`, `sub`,..., providing all "shared behaviour" for the latter.

1.4 Restrictions of the Machine Model

No explicit provisions are made for the description of

- self-modifying code

- i/o devices
- interrupts
- the underlying operating system
- sub-instructions and multi-cycle-instructions.

These may be described ‘by hand’. E.g., operation system calls may be modelled as instructions:

```
op openfd()
syntax="move #42,d0; trap 1"
action={canonical("fopen",a0,a1);}
```

Of course, much more elegant ways to describe these concepts may be added easily, but they would complicate the semantics a lot while not being very general. `nML` in its present form is a core language that can be extended at need.

2 Syntax and Semantics of the `nML` Attribute Grammar

A `nML` description is a file consisting of an attributed instruction grammar[3] and assorted definitions.

2.1 General Description of Attribute Grammars

A *context-free grammar* G is a 4-tuple $G = (N, T, P, S)$ consisting of *nonterminals*, *terminals*, *production rules* and a *start symbol*. A *token* is a terminal or a nonterminal. The set S_G of *Strings* in G is the set of all sequences of tokens, i.e $S_G = (N_G \cup T_G)^\star$. Traditionally, $\alpha, \beta, \ldots$ denote strings. N, T and P are finite; $N \cap T = \emptyset$, $S \in N$, $P \subset N \times S_G$.

A string t may be *derived in one step* from a string s, written $s \xrightarrow{1} t$ iff

$$s \equiv \alpha x \beta \ \wedge \ t \equiv \alpha \gamma \beta \ \wedge \ (x, \gamma) \in P$$

A string t may be *derived in n steps* from a string s, written $s \xrightarrow{n} t$, iff $\exists u : s \xrightarrow{1} u \ \wedge u \xrightarrow{n-1} t$. t *is a derivation of* s ($s \xrightarrow{*} t$) if it may be derived in a finite number of steps.

A string α is *terminal* if it consists only of terminals, i.e. iff $\alpha \in T^*$. A grammar is *acyclic* if there exist no nonterminal x with a derivation $x \xrightarrow{*} \beta x \gamma, |\beta\gamma| > 0$.

The language $L(G)$ of a grammar G is the set $\{\alpha | \alpha \in {T_G}^* \wedge \ S_G \xrightarrow{*} \alpha\}$.

Acyclic grammars have finite languages.

An *attribute grammar* is a grammar in which for each nonterminal a fixed set of attributes, and for each production a set of *sematic rules* is given. For a given derivation, the semantic rules determine the values of the attributes. A theory of attribute grammars is given in [10]. [4]

2.2 `nML` grammars

For `nML` grammars, all nonterminals have to have derivations. There may be no cycles. As a consequence, all strings that have no productions are terminal.

A `nML` grammar description differentiates between two subsets of N, $N_\wedge$ and $N_\vee$, and two sets of production rules $P_\wedge$ and $P_\vee$. $N_\wedge \cup N_\vee = N, N_\wedge \cap N_\vee = \emptyset, P_\wedge \cup P_\vee = P, P_\wedge \cap P_\vee = \emptyset$.

$P_\wedge$ is the set of production functions $N_\wedge \to T_G$; while $P_\vee$ is the set of production relations $P_\vee \subset N_\vee \times N$. $P_\wedge$ models *and-rules*, while $P_\vee$ models *or-rules*. In an `nML`-grammar, there may be no 'mixed' rules.

Semantically, each terminal string produced by the grammar corresponds to one instruction in the instruction set. By itself, such a string contains no useful information: the instruction's syntax and semantics are defined by the string's attributes. For each attribute, its semantic function is given, i.e. the attribute declaration declares the attribute and its definition in one step.

Textually, a production in $P_\vee$ looks like

[3] First described by Knuth[11] in 1965.

[4] Full-fledged attribute grammars know two kinds of attributes: *synthesized* and *inherited*. If both occur together unrestricted, attribute evaluation can become quite expensive. For `nML` applications, synthesized attributes should suffice, so inherited attributes are silently ignored.

```
op n0 = n1 | n2 | n3 | ...
```

while a production in $P_\wedge$ looks like

```
op n0(p1:t1,p2:t2, ... )
a1 = e1  a2 = e2 ...
```

where each `n`*i* is a nonterminal and each `t`*i* is a token. Each `a`*i* is an attribute name, the `e`*i* their respective definitions. The `p`*i* are just names for the parameters to be used in the attribute definitions.

Productions in $P_\vee$ have no attribute definitions; Nonterminals in $N_\vee$ simply pass the attributes through.

The start symbol is fixed to be the identifier `instruction`.

Attributes have *expressions* as their definitions. Expressions are arbitrary C-like expressions or sequences of statements. Expressions may contain references to attributes of parameters. An attribute reference *param.attr* refers to the value of the attribute *attr* of the parameter *param*.

What follows is a complete, if not very interesting, `nML` grammar:

```
op instruction(f:foo,g:bar)
size=f.size+g.size

op foo()
size=1

op bar()
size=2
```

this enumerates one 'instruction' , which has the one attribute `size` with value `3`. In contrast, the grammar

```
op instruction(f:foo,g:barOrbaz)
size=f.size+g.size

op foo()
size=1

op barOrBaz = bar | baz

op bar()
size=2

op baz()
size=3
```

enumerates two 'instructions' of `size`s `3` and `4`.

3 The Pre-Defined Attribute Set

Three attributes are pre-defined: `syntax`, `image` and `action`.

The `syntax`-attribute describes the textual (=assembler) syntax of the instruction; it has to evaluate to a string.

The `image`-attribute describes the binary coding of the instruction, it has to evaluate to a *binary string*, which is a string containing only `1`s, `0`s and whitespace. The latter is ignored.

The `action`-attribute describes the semantics of an instruction; it has to evaluate to a sequence of register-transfer operations. The exact syntax of the latter is shown in section 4.

What follows is a short grammar that is complete in regard to the three pre-defined attributes.

```
type addr=card(24)
type long=card(32)
mem M[2**24,long]
mem PC[1,addr]

op instruction = jump | binop

op jump(a:addr)
syntax=format("jump %d",a)
image=format("1000 0000 %24b",a)
action={ PC=a; }

mem tmp1[1,long]
mem tmp2[1,long]

op binop(x:binaction,a1:addr,a2:addr)
syntax=format("%s %d,%d",x.syntax,a1,a2)
image=format("11%6b %24b %24b",x.image,a1,a1)
action= {tmp1=M[a1];
         tmp2=M[a2];
         x.action;
         M[a2]=tmp2;
}

op binaction= plus | move

op plus()
syntax="add"
image="00000"
action={ tmp2=tmp1+tmp2; }

op move()
syntax="move"
image="00001"
action={ tmp2=tmp1; }
```

This little grammar enumerates three instructions (or better: instruction templates).

The first of these has one terminal argument 'a' of type `addr`, i.e. `card(24)`. It describes the instruction `jump` x, where x may be any number between 0 and $2^{24}-1$.[5] The semantics of the jump-instruction is described as the *sequence* containing the one assignment-statement `PC=a;`. The register `PC` has special semantics: it is assumed that during the execution of any instruction, `PC` holds the address of the instruction to be executed next. So, changing `PC` enables an instruction to jump somewhere else, where by default the instructions in a program are executed sequentially. This is, indeed, the only way to manipulate control flow.

The other two instructions, `add` and `move`, have a common description for fetching and storing arguments. In the case of the move operation, there is an 'unnecessary' load operation (`tmp2=M[a2]`), which is, however, semantically irrelevant. One can see how auxiliary registers are introduced to facilitate code sharing. As a side-effect, the physical data-path is re-modeled. These auxiliary registers don't have any impact on the functional model of the machine, because they cannot 'carry state' from one instruction to the next – at least not in the way they are used now.

3.1 Addressing Modes

`nML` supports the concept of addressing modes. Suppose the mode declarations

```
mem A[8,long]
mem D[8,long]

mode SRC = IMMS | IMMW | IMML | REG | IND | INDOFFSET

mode IMMS(n:int(8))=n
syntax=format("#%d",n)
image=format("%8b",n)

mode IMMW(n:int(16))=n
syntax=format("#%d",n)
image=format("%16b",n)

mode IMML(n:int(32))=n
syntax=format("#%d",n)
image=format("%32b",n)

mode REG= AREG | DREG

mode AREG(n:card(3))=A[n]
syntax=format("A%d",n)
image =format("%3b",n)

mode DREG(n:card(3))=D[n]
syntax=format("D%d",n)
```

[5]It is important to see that this grammar does *not* descibe $2^{24}+2*2^{24*2}$ different instructions, but only three: to the grammar, all terminals/types are equally opaque.

```
image =format("%3b",n)

mode IND(R:AREG)=M[R]
syntax=format("(%s)",R.syntax)
image=R.image

mode INDOFFSET(R:AREG,O:DREG)=M[R+O]
syntax=format("(%s,%s)",O.syntax,R.syntax)
image=R.image
```

This is an incomplete subset of the addressing mode grammar of a 68000. (Incomplete, because no way exists to distinguish between modes. In reality, special marker bits would be provided by auxiliary attributes.)

One can see that the difference between addressing modes and 'normal' grammar rules ('op' rules) is the existence of a 'value'. E.g, the `AREG` rule has the value `A[n]`, while the `INDREG` mode has the (composed) value `M[A[n]]`. Addressing modes are used as follows:

```
op add(src:REG,dst:REG)
action={ dst=src+dst; }
```

Now, if `src` is an `AREG` and `dst` is a `DREG`, this is the same as

```
action={ D[n]=A[m]+D[n]; }
```

for some values of `n` and `m`. That is, a parameter that stands for an addressing mode is replaced by its 'value'.

This can be modelled by a special attribute `value`. Simply imagine all mode declarations transformed into declarations like

```
...
op INDOFFSET(R:AREG,O:DREG)
value=M[R.value+O.value]
```

and all uses of modes into

```
...
action={ dst.value=src.value+dst.value; }
```

3.2 Type declarations

In addition to the instruction grammar, a `nML` description contains declarations for memory objects, data types, constants and macros.

A *data type* describes a set of values, e.g. the type `card(8)` describes the set of numbers 0...255. In the grammar, data types are used as terminals (they could as well be seen as grammar rules, i.e. card(8) could denote 256 expansions, but this would blow up the grammar without need). `nML` knows about the following type constructors:

- `int(`n`)`
  is the type of n-bit signed numbers in 2s-complement representation.
- `card(`n`)`
  is the type of n-bit unsigned numbers.
- `float(`n`,`m`)`
  is the type of floating-point numbers with n bit mantissa and m bit exponent. While no provision for NaNs and infinities are made, a IEEE-754 representation may be assumed.
- `fix(`n`,`m`)`
  is the type of signed fixed-point numbers with n bits before and m bits after the binary point.
- `[`n`..`m`]` (where $n \leq m$)
  is the type of (integer or cardinal) numbers in the range of $[n \ldots m]$.
- `enum(`id_1`,...,`id_i`)`
  defines an enumeration type, i.e. the type card($\lceil \log_2(i) \rceil$) and the constants id_1=0, ..., $\mathrm{id}_i = i - 1$.
- `bool`
  denotes the boolean values. Two constants `true` and `false` are pre-defined. If coerced to an integer, `true` has the value `-1`, while `false` has the value `0`. Wherever an integer is needed (essentially, only in `if`-expressions), a `0` will be interpreted as `false`, while everything else will be interpreted as `true`.

A *type definition* like

```
type byte=card(8)
```

defines a synonym for a type expression.

3.3 Memory declarations

A *memory declaration* like

```
mem A[8,card(32)]
```

defines a *memory base*, i.e. a set of memory *locations* accessible under a name and an index. A location is a place where an value of a type may be stored. E.g., the above shown declaration introduces a memory base called `A` that contains 8 locations, denotable as `A[0]`...`A[7]`, which may be used to store numbers in the range of $0 \ldots 2^{32} - 1$. Memory bases and locations are *not* terminals of the grammar; in fact, they don't exist in the grammar at all, only in the `action` attribute definitions.

Memory declarations can have additional attributes:

```
mem M[2**32,byte] alignment=2
```

declares an alignment restriction (if these are supported; they are *not* part of the core language);

```
mem A[8,int(32)]
mem SP[1,card(32)] alias=A[7]
```

declares SP to be an alias of A7, i.e. both denote the same location, but they have different type interpretations;

```
mem PORT1[1,byte] volatile="port1" alias=M[0xffffff84]
```

declares a memory location to be "volatile", i.e. able to change at random. The value of the volatile attribute may hold additional information for the entity that reads the description.

There is no predefined attribute for marking memory bases as 'temporary' – in the sense of the registers tmp1 and tmp2 in the first example – , because this property can be deduced automatically.

One last attribute is the program_memory declaration. A declaration of the form

```
mem M[32000] program_memory
```

declares the memory base M to be the one holding programs. Per default, the largest memory base is assumed to hold the programs.

When modelling machines where memory is divided into different purpose parts, one can combine the program_memory and the alias attributes:

```
mem MEM[2**24,byte]
mem PROGMEM[2**20] program_memory alias=MEM[1024]
```

declares the PROGMEM as part of the MEM memory base, starting at the address 1024.

The special memory base of size 1 pc (or PC) holds the program counter. There has to exist a PC variable in every machine. Assignment to PC means changing the program counter and thereby choosing a different next instruction. During the execution of an instruction, the PC points to the next instruction to be executed. By writing to the memory pointed to by the PC, one could theoretically write self-modifying code. *THIS IS NOT SUPPORTED*, i.e. tools do not have to model this behaviour faithfully.

The PC interacts with the special optional global parameter pipeline_factor that determines the number of jump delays slots.

3.4 Constants and Global Parameters

A declaration like

```
let A=100
```

declares a global constant A to have the value 100. Such a constant might be used in every context its value could stand. Any constant may be defined only once.

Constants may be used to extend nML: Any information about a machine that can be given with a single number or string can easily be defined as a constant (with a default value, so that standard nMLdescriptions still work).

In core nML, there is just one such constant (or "global parameter").

This is the pipeline_factor. On machines with an instruction pipeline visible to the programmer, there are *delay slots* whenever a jump occurs. Usually, there is one such slot, but two are not unheard of. A declaration

```
let pipeline_factor=1
```

introduces one delay slot after each instruction that changes the program counter. The default value is 0.

3.5 Macros

A macro-definition like

```
macro max(A,B)= if (A)>(B) then A else B endif
```

defines a pseudo-function. Macros may not introduce circularities, neither direct nor indirect! They are of no further interest, because a simple syntactic expansion can remove them painlessly.

3.6 execs

An optional functionality introduced in version 1.3 is that of *exec*s. An exec is a memory location that stores not a run-time value, but a *behaviour* – an exec variables holds as its value a sequence. Execs make it possible to model pipelining behaviour, delayed writes, and other unpleasant aspects of architectures of the more crufty persuasion.

The only operations defined on exec values (i.e. sequence) are store, fetch, and execute (a new primitive function) – hence the name. Exec locations are assignment-incompatible with non-exec locations. Exec locations have no size. An example: delayed branch:

```
mem branch_slot[1,exec] init = {}
...
op instruction(a:rest_op)
action = { exec(branch_slot);
           branch_slot={}
   a.action;
         }
...
op branch(dst:word)
action={
        branch_slot={PC = dst}
       }
```

i.e., the PC will be re-set as part of the next instruction. To make this well-defined, an initial setting of the branch_slot with a no-op ({}) has to be done.

The value stored in the exec location has to encode two informations: the action to be performed and the formal parameter values (in this case, the value of “a”). In this respect, an exec value is like a function closure.

Since `execs` are not suited for code generation, memory latency annotations have been introduced in version 1.5. From then on, `execs` are considered as “optional feature” that is *not* part of “standard” nML!

4 Attribute Expression Syntax and Semantics

An *expression* is a term that can be evaluated to a value. A value is either a logic value, a number, or a string. Expressions are used both to compute values of attributes and as parts of register transfer sequences in action attribute values. These really are two different uses of the same expression syntax and semantics; this dual use leads to restrictions in the set of expressions allowed as direct values of attributes.

4.1 Expressions

An expression is either

- a *constant* like `13` or `"add %s,%4d"`. Numeric constants may be written to base 2 or base 16, as in `0b00100010` and `0x12ab`,
- an *attribute reference* like `arg1.syntax`,
- a *memory location* like `PC` or `M[12]` or `A[D[x-1]+4]`, containing the name of a memory base and an arbitrary indexing expression (Location bases of size 1 can be accessed without an index. Examples are the program counter and single condition bits.),
- a *function call* like `a+b` or `format("%s",a.x)`,
- a *macro application* like `MAX(a,b)` where `MAX` is defined by a macro definition like
  `macro MAX(A,B) = if (A)>(B) then A else B endif`
  Such a macro application can be evaluated by replacing it textually by its definition in the 'obvious' way.
- a *conditional* like `if a>b then x else y endif`, which returns the value of the evaluated expression,
- a *switch expression* like

  ```
  switch x {
   case 0: "load"
   case 1: "store"
   default:"move"
  }
  ```

  that evaluates to the one selected value (the selection has to be exhaustive!).

There is a list of predefined functions and operators:

- `+,-`
  these are the usual arithmetic functions. Applied to two numbers of type X, they return type X. In the case of Integer(N) or Cardinal(N) arguments, all functions are defined modulo 2^N. Applied to Integer(N) and Integer(M) arguments, the result is Integer(max(N,M)). The same applies for different-sized Cardinal arguments. In the case of Cardinal(M) and Integer(N)

arguments, the result is of type Integer(max(N,M))[6]. In the case of floating point or fixed point arguments, both argument types have to be the same.

- `*`,`/`,`%`
  these are the usual multiplication, division and remainder functions. In the case of Integer or Cardinal arguments, the same rules apply as for `+` and `-`. In the case of floating point or fixed point arguments, mixing with integers and cardinals is allowed, the result type being that of the float or fix argument.

- `**`
  is the integer exponentiation function. The second argument has to be a constant.

- `>`,`<`,`>=`,`<=`,`==`,`!=`
  The standard numeric comparison functions. These may be applied to all kinds of numbers. They return a boolean value, which is equivalent to either `-1` (true) or `0` (false).

- `<<`, `>>`, `&`, `|`, `^`
  the binary shift and mask functions from C. These may be applied to Integer, Fixpoint and Cardinal values, only. If applied to a Fixpoint(n,m), the latter is seen as a Integer(n+m), i.e. no shifting is done.

- `<<<`, `>>>`
  Rotate right and left, defined on cardinals and integers.

- —mant—, `exp`
  Select the mantissa and exponent of a floating point number. These are represented as integers of sufficient size, i.e. a bias-128 exponent will be represented as 8-bit integer.

- —mkfloat—
  Create a floating point number from a mantissa and an exponent.

- `not`
  The logical not. Delivered to any non-floating point value, returns -1[7] if the value is 0 and 0 if its non-zero. The result type is the same as the argument type.

- `&&`, `||`
  The logic functions from C. They accept locic values and integers (0 is false) and deliver logic values.

- `~` The binary complement. Delivered to any non-floating point value, returns a number of the same type that has all bits reverted.

- `&`, `|`, `^`
  The binary functions from C. They accept non-floating point values as arguments. The result type has the type of the longer argument; the shorter argument is filled up with zeros or sign-extended, respectively.

- `coerce(`*type-expr*`,` *value*`)`
  this function, when applied to any numeric value, delivers the "best approximation" of

[6] An overflow is possible. 2s-complement is assumed.
[7] 2s-complement for 'all 1s'.

the value in the type to be coerced to. When coercing signed to unsigned, 2s-complement representation is assumed. When coercing from floating or fixed point numbers to integer, everything behind the binary dot is cut off.

- `cast(`*type-expr*`,`*value*`)`
  Re-Interprets the *value* as an object of type *typename*. The original type of *value* has to be of the same bit width as *typename*.
- `bits(`*value*`)` Returns the number of bits of the type of *value*. The return type is a suitably large cardinal (say, 8 or 16).
- `canonical(`*string*`,`*args...*`)`
  or
  `"`*string*`"(`*args...*`)`
  this function applies a function of unknown semantics. It may be used in `action` attribute definitions only. To give an example of its use: a machine that directly implements trigonometric functions will need a register transfer like
  `dst=canonical("sin",src)`
  It is assumed that the entity that reads the description knows what is meant by the canonical function. Canonical functions may only be used as 'objects' in semantic attributes; they *must not* be used in computing the attributes themselves!

  The quote-syntax has been introduced in version 1.5.
- `format(`*format-string*`,`*args...*`)`
  This function is used to put together the string values of the `syntax` and `image` attributes. The format string is a variation of the `printf` format string well known from C. It may contain alphanumeric characters, blanks, tabs ('`\t`'), newlines ('`\n`'), and format directives of the form `%`*nC*, where *n* is an optional field size and *C* is one of the following characters:
  - `d`
    This takes an Integer or Cardinal argument from the argument list and formats it as a decimal number.
  - `b`
    This takes a Cardinal argument and formats it as a binary number. It may also take a binary string, i.e. a string containing only `1`s, `0`s and (ignored) whitespace.
  - `x`
    This takes a Cardinal argument and formats it as a sedecimal number.
  - `s`
    This takes a string argument and incorporates it as a whole.

4.2 The Type of Constants

The type of a constant is assumed to be "of infinite precision". E.g., the constant `3` denotes the "ideal" cardinal 3. Operations on constants preserve this. E.g., the result of `1/3` is the 'ideal' value 1/3. In a situation like

```
A0 = D0 * (1/3) + D1 * (2/3)
```

(which is quite unlikely to occur), the intermediate computation is ideal, i.e. only on storing to `A0` is any rounding done.

4.3 The `error` function

The pseudo-function "`error`" models the behaviour of the machine on encountering an illegal state. Calling `error` (with an optional string argument that describes the error) results in a totally unspecified machine state. A simulator should abort the execution upon encountering `error`; a compiler should try to avoid generating code that calls `error`.

4.4 The `undefined` function

The pseudo-function "`undefind`" creates an undefined value. Applying the function to a type will create a undefined value of this type. Using `undefind`, nondeterministic behaviour can be modelled, as in:

```
action={...
if (undefined(bool)) then ...
                             else ...
        ...}
```

The creation of an undefined value does not constitute an error!

4.5 Multiple Return Values

As of version 1.5, canonicals can have multiple return values. An example of the syntax is

```
mem Ci[1,bool]
mem Co[1,bool]
mem R[16,int(16)]
action={...
(R[1],Co) = "addc"(R[0],R[1],Ci)
        ...}
```

4.6 Standard Idioms for overflowing bits

One problem often encountered is that of overflowing bits, i.e. operations that "push out" bits on the "border". The standard idiom to represent these bits employs the bit-concatenation operator ("::"). For example, to catch a bit that has been "shifted out" of a word, one could write:

```
mem X[1,bool]
mem R[16,int(16)]
action={...
X::R[0] = R[1]<<1
        ...}
```

In this context, the shift-right operator will generate a 17-bit result, which is then splitted into a single bit and the 16-bit main result.

4.7 Sequences

The `action` attribute has *register-transfer sequences* as value. Such a sequence is built up from *statements*. Textually, a sequence is enclosed by braces ({ and }); each statement in a sequence is delimited by a semi-colon (;).

A statement is either

- an *assignment* like `a=b+c`, where the result of an arbitrary expression is assigned to a location.
- a *conditional statement*, which looks like a conditional expression, but which contains two sequences instead of two expressions,
- a *switch statement*, which looks like a switch expression, but which contains sequences instead of expressions.

Sequences may contain calls to canonical function. Expressions occuring in sequences may refer to locations.

4.8 Coercion rules for assignment statements

The only kind of action done by a register transfer sequence is that of assigning values to locations. `nML` provides coercing rules for assignments between locations of different types.

There is one main rule: assignment between locations of *equal size* is a direct, un-coerced operation. Assignment between locations of *different size* is either done with a coercion, if the types are compatible, or not allowed.

Assume the definitions

```
type byte=card(8)
type sbyte=int(8)
type long=card(32)
type slong=int(32)
type float32=float(24,8) \ 24 bit mant., 8 bit exp.
mem M[2**32,byte]
mem D[8,long]
mem F[8,float32]
```

in which five data types and three memory bases are defined. Let's look at some statements using these definitions:

```
M[100]=M[101];
```

This simplest possible case moves a 'byte' from one location to another. No coercion or casting takes place.

```
D[0]=F[0];
```

Here, a ‘float32’ value is moved into a location that is tagged as ‘long’. Since both locations have the same size (32 bits), the value is moved regardless of the incompatibility of types.

```
        D[0]=M[100];
```

Here, a ‘byte’ is taken and put into a ‘long’ register. An implicit coercion takes place, i.e. what really happens is:

```
        D[0]=coerce(long,M[100]);
```

In the case of coercion between signed and unsigned values, as in

```
        mem SB[1,sbyte]
         ...
        D[0]=SB[0];
```

presumably sign-extension is done. This is *not* guaranteed! To have guaranteed sign extension, use

```
        mem SB[1,sbyte]
         ...
        D[0]=coerce(slong,SB[0]);
```

Here, a signed byte is coerced (=extended) to a signed long; then the “equal size” rule takes charge and puts the value unchanged into the unsigned location.

Lastly, something like

```
        F[0]=M[100];
```

is not allowed.

4.9 Bit-Fields

To address a sub-field within a memory location, the syntax *location*<*left*..*right*> may be used. For example, to shift a 32-bit register on bit to the left, with the lsb staying the same, one can use

```
  R[0]<1..31> = R[0]<0..30>
```

Subfield selection within an expression (i.e., on the “right side” of an assignment) returns an *integer* of the given size. Applied on the right side of an expression, a subfield specifies a cardinal location of the indicated size. The part of the register that is not specified in the selection (in the example, bit 0) stays unchanged.

The index 0 means the lsb, i.e., in integer numbers, the bit with the value of 1. Negative indices and indices that are bigger than the field are forbidden.

4.9.1 Bit Reversion using Bit-Fields

The normal order of the indices is <*lsb*..*msb*>. But, one can assume that the intuitive semantics of

```
R[0]<0..31> = R[0]<31..0>
```

is that of bit reversal.

4.9.2 The format operator and Bit fields

One can combine bit fields with the format operator. For example, to switch the byte ordering of a long number from "1234" to "4321", one can say

```
R[0] = format("%b%b%b%b",R[0]<24..31>,R[0]<16..23>,R[0]<8..15>,R[0]<0..7>,)
```

The format operator, used for bit strings only, returns a string that can be interpreted as a number.

This usage is the reason for defining "Integer" as the return type of bitstring selection, so that, e.g, a "16-bit sign-expand to 32-bit" can be written down at

```
R[0]<0..31> = format("%32b",R[0]<0..23>)
```

(Which could, otherwise, have been written as

```
R[0]<24..38> = format("%8b",R[0]<23..23>)
```

or, quite simple (and tricky),

```
R[0] = R[0]<0..32>
```

)

4.10 Concatenation of Values

The `::` operator allows the concatenation of arbitrary expressions; it is defined on the left side of assignments, too (if the expressions denote memory locations only, of course). For example,

```
M[0] :: M[1] = R[4]
```

assigns the value of R[4] to the locations M[0] and M[1]. The order is the same as used in sequences of locations, i.e., essentially undefined.

4.11 Assignment to Sequences of Locations

On most machines, addressing is per byte, while registers hold multi-byte values. Under declarations

```
mem M[2**32,byte]
mem D[16,long]
```

an assignment like

```
M[100]=D[0];
```

means to store the (long) value of D0 into the sequence of byte locations M[100]...M[103]. While the above sequence is *not* allowed in pure `nML`, a simple extension could be defined as follows: when a global constant `byte_order` is set to one of the strings `"big"` or `"little"`, the above statement is defined when the size of the destination is a multiple of the size of the source (It is still ill-defined to store a 30-bit value into a byte array). The semantics is that the source value is split up and distributed over the indices (in this case, 100...103). The order is 'big-endian' (most significant byte first), if byte_order is set to "big", and 'little endian' otherwise.

A first problem arises: The semantics of

```
D[0]=M[100]
```

should now be changed in a symmetrical way to be that of

```
D[0]=M[100..103]
```

(which is, of course, unsyntactical). To load a byte into a long location under this changed semantics, a temporary must be introduced:

```
mem TMPBYTE[1,byte]
TMPBYTE=M[100];
D[0]=TMPBYTE;
```

A second problem is that of bounds: lets assume an assignment to the top of memory:

```
M[2**32-2]=D[0]
```

On most machines, this will 'swap over' to addresses 0 and 1. What happens if the memory has a size different then 2^N? Assume

```
mem X[1200]
X[1198]=D[0]
```

This could cause a trap or wrap around into some totally unexpected place.

A third problem is that of alignment: on aligned machines, the machine description has to provide the information that an instruction like

```
M[1]=D[0]
```

will cause an alignment exception. This information can be given as a memory attribute like

```
mem M[2**32,byte] alignment=2
```

Or, when different sizes have different alignments:

```
mem M[2**32,byte] alignment=true
```

which could define that values of size 2^{N*8} are aligned on addresses that have the last $N-1$ bit cleared. The latter would, as a side effect, solve the problem of wrap around at the end of memory (when the memory is of an aligned size, but any architecture missing *this* constraint would be *truly* weird!).

As said before: multi-word memory access is *not* part of the core `nML` language.

4.12 Problems with Aliases

Originally, aliases were only introduced to be able to have "symbolic names" for registers (`PC`,`SP`) or parts of memory (`ZEROPAGE`,`INTERRUPTVECTORS`). Later, the esteemed first tester introduced the trick of mapping a register to a bit array to get at the msb and lsb directly. Now, while this was not intended, it proved to be good, so it was not explicitely forbidden in later versions of this report. But problems still exist. The main one is the order of bits in memory (just the same problem as with multi-word memory access, really): given declarations like

```
mem M[2**10,long]
mem MBITS[bits(long)*(2**10),bool]
```

what will an action like

```
M[0] = 0x789a
MBITS[0]=MBITS[1]
```

result in? In different machines, `789a` may be represented in memory directly, or as `9a78`, or as `a987`, or as `87a9`.

As it stands now, the above given action sequence is not well defined. And this is, in my opinion, the best way. It is trivial to introduce ad-hoc parameters like

```
let byte_order = "3412"
```

to cope with any such problem when it occurs, but there is no simple general way to solve this nasty little problem. It will usually be much simpler to just avoid any such ambiguities by writing clean definitions.

5 Pipelines

In version 1.5, the pipelining model – which consisted of the sole `pipeline\_factor` in previous versions – has been extended and specified.

5.1 Pipelined Execution Model

5.2 Memory Latency

5.3 The `next` Pseudo-Function

5.4 The Pipelined `image` Attribute

5.5 Default Settings

6 The Do's and Don'ts of nML

Practice has shown how easy it is to write machine specifications that are hard to understand both for a human reader and for an analysis program. The following section shall establish a few guidelines for writing nML descriptions.

6.1 Top-Down, Bottom-Up design

6.2 The "inner logic" of an instruction set

6.3 Fancy images

6.4 Use of canonicals

6.5 The danger of overspecification

6.6 Alias Madness

die von georg gebauten konstruktionen....alias auf alias auf alias -¿ ideal: alias nur von groesseren auf kleinere; keine element-uebergreifenden aliase

6.7 When to use `undefined`

6.8 Errors

6.9 How do I model ...?

7 A complete nML description

In the following, a complete nML description of a simple RISC-like machine is given. While being quite small, additional complexity is introduced through complex addressing-modes.

```
\ small.m --- description of a small, fictional machine

let REGS=4               \ 2^4 registers

type word=card(16)
type long=card(32)
type index=card(REGS)    \ register index type

mem M[2**32,long]        \ main memory
mem R[2**REGS,long]      \ registers

mem CZ[1,bool]           \ condition code
mem CN[1,bool]           \ bits

mem PC[1,long]           \ program counter

\ 2 kinds of addressing modes: short (5 bit) and long (7 bit)
\ short:
\ name Image    Syntax
\ MEM  0nnnn    (Rn)
\ REG  1nnnn     Rn
\ long:
\      00<short>
\ IMM  1iiiiii   #x    ,in the range -32...31
\ INC  010nnnn  (Rn)+
\ DEC  011nnnn  -(Rn)
\ post-increment and pre-decrement are modelled by attributes

mode MEM(i:index)=M[R[i]]
syntax=format("(R%d)",i)
image=format("0%4b",i)

mode REG(i:index)=R[i]
syntax=format("R%d",i)
image=format("1%4b",i)

mode SHORT = MEM | REG

mode LSHORT(s:SHORT) = s
syntax=s.syntax
image=format("00%b",s.image)
pre={}       \ these dummies have to be inserted to
```

```
post={}      \ make the attributes defined for all LONGs

mode IMM(n:int(6))=n
syntax=format("#%d",n)
image=format("1%6b",n)
pre={}
post={}

mode PRE(r:MEM)=r
syntax=format("-%s",r.syntax)
image=format("010%4b",r.image&0b1111) \ remove tag bit
  \ the removal uses the fact that bit strings are just
  \ numbers with a field size, so arithmetic operations
  \ like masking can be used on them
pre={ r=r-1; }
post={ }

mode POST(r:MEM)=r
syntax=format("%s+",r.syntax)
image=format("011%4b",r.image&0b1111) \ remove tag bit
pre={ }
post={ r=r+1; }

mode LONG = LSHORT | IMM | PRE | POST

op instruction(x:instr_action)
action={           \ these are the actions done in
                   \ each instruction
        R[0]=0;    \ R0 holds 0 constantly
        x.action;  \ here the different actions are inserted
       }
syntax=x.syntax
image=x.image

op instr_action = control_op | alu_op | move_op

op control_op = test_op | branch_op
              | jsr_op | rts_op

op test_op(src1:LONG,src2:SHORT)
action={
        src1.pre;
        CZ=src1==src2;
        CN=src1<src2;
        src1.post;
       }
syntax=format("cmp %s,%s",src1.syntax,src2.syntax)
```

```
image =format("0000 %b %b",src1.image,src2.image)

type testcode = enum(tr,         \ true
                     zc,zs,      \ CZ clr/set
                     nc,ns)      \ CN clr/set

op branch_op(newpc:LONG,code:testcode)
action={
        newpc.pre;
        if code==tr
        ||(code==zc && CZ==0)
        ||(code==zs && CZ!=0)
        ||(code==nc && CN==0)
        ||(code==ns && CN!=0)
        then PC=newpc;
        endif;
        newpc.post;
       }
syntax=format("b%s (%s)",switch(code){
                                 case tr: "ra"
                                 case zc: "eq"
                                 case zs: "ne"
                                 case nc: "mi"
                                 case ns: "pl"
                                },newpc.syntax)
image =format("0001 0%3b %b",code,newpc.image)

op jsr_op(nextpc:LONG,link:SHORT)
action={
        nextpc.pre;
        link=PC;
        PC=nextpc;
        nextpc.post;
       }
syntax=format("jsr (%s),%s",nextpc.syntax,link.syntax)
image =format("0110 %b %b",nextpc.image,link.image)

op rts_op(link:LONG)
action={
        link.pre;
        PC=link;
        link.post;
       }
syntax=format("rts (%s)",link.syntax)
image =format("0111 %b",link.image)

mem SRC1[1,long]        \ temporary registers
mem SRC2[1,long]
mem DST[1,long]

op alu_op(src:LONG,dst:SHORT,aa:alu_action)
action={
        src.pre;
        SRC1=src;
        SRC2=dst;
        aa.action;
        dst=DST;
        src.post;
       }
syntax=format("%s %s,%s",aa.syntax,src.syntax,dst.syntax)
image =format("1%b %b %b",aa.image,src.image,dst.image)

op alu_action= a_add | a_sub | a_and | a_or | a_mult | a_div | a_rem

op a_add()
action={ DST = SRC1 + SRC2; }
syntax="add"
image="000"

op a_sub()
action={ DST = SRC1 - SRC2; }
syntax="sub"
image="001"

op a_and()
action={ DST = SRC1 & SRC2; }
syntax="and"
image="010"

op a_or()
action={ DST = SRC1 | SRC2; }
syntax="or"
image="011"

op a_mult()
action={ DST = SRC1 * SRC2; }
syntax="mult"
image="100"

op a_div()
action={ DST = SRC1 / SRC2; }
syntax="div"
image="101"
```

```
op a_rem()
action={ DST = SRC1 % SRC2; }
syntax="rem"
image="110"

op move_op = move | store | lconst | sconst

op move(src:LONG,dst:SHORT)
action={
        dst=src;
       }
syntax=format("move %s,%s",src.syntax,dst.syntax)
image =format("0010 %b %b",dst.image,src.image)

op store(src:SHORT,dst:LONG)
action={
        dst=src;
       }
syntax=format("move %s,%s",src.syntax,dst.syntax)
image =format("0011 %b %b",src.image,dst.image)


op lconst(dst:REG,value:long)    \ the only >1-word-instruction
action={
        dst=value;
       }
syntax=format("move #%d,%s",value,dst.syntax)
image =format("0100 %b 0000000 %b",dst.image,value)

op sconst(dst:SHORT,value:int(7))
action={
        dst=coerce(int(32),value);
       }
syntax=format("moveq #%d,%s",value,dst.syntax)
image =format("0101 %b %b",dst.image,value)
```

8 Changes from Previous Versions

Changes between version 2.0 and version 1.5:

This version introduces the hardware modelling cell (HMC) grammar and its related concepts.

- (TODO) HMC grammar: `cell` keyword, `opr` keyword
- (TODO) Introduction of the `activate` attribute
- (TODO) Pipelining semantics; re-definition of `pipeline_factor`.
- (TODO)

Changes between version 1.5 and version 1.2:

- (TODO, Andi meint, es waer "verworfen") new operator `sign`. Takes one argument, gives the sign (0=positive, 1=negative)
- (TODO) syntax-change: switch ... [case ...] default ... end
- (TODO) syntax-change if ... then ... else ... end
- (TODO) syntax-change: "coerce(type,expr)" can now be written as "type(expr)" werden
- (TODO) Clarification: "mem[addr]¡from..to¿" is the same as "mem[addr]¡to..from¿"
- (TODO) Extension: "mem[addr]¡from..to¿" with from and/or to negative: addresses from the front. -1 is the MSB, -2 the second-most significant bit, -(wordlength) the LSB.
- (TODO) removed: delayed evaluation via `exec` locations. (Too complex)
- (TODO) pipelining model that employs latency annotations for memory locations.
- (?) multi-cycle instructions via `next();` canonical
- (?) new image separator `;` to indicate load latencies
- (TODO) more precise semantic definition for "`+`", "`-`", "`*`", "`/`", "`%`", "`<<`", "`>>`"
- (TODO) definition of standard idioms for carry and overflow extraction
- new pseudo-functions: `undefined(type)`, `error()`
- new operators: "`>>>`" (rotate right), "`<<<`" (rotate left)
- new floating-point operators: "`mant`", "`exp`", "`mkfloat`"
- (?) operator attributes
- (?) statement attributes
- (?) optional net list declarations

- (?) timing declarations as attributes to net list identities
- (?) simplification of casting system (cast/coerce and explicit/implicit rules); bit alignment
- introduction of "encoding" attribute for types. introduction of encoding-cast rules etc.
- (?) general boolean primitives specified as tables
- (?) general bit-connect primitives specified as tables
- (?) new literal type: predefined lookup-table
- (?) extra-nML: convention for defining the semantics of canonicals as a C library intra-nML: canonical attributes?
- simplified syntax for canonicals (`"sin"(x)` instead of `canonical("sin",x)`

The following will be probably be part of the version 2.0:

- (?) multiple-output operations
- (?) idioms for multiple-sized processors (byte/word/long instructions)

Changes between version 1.3 and version 1.2:

- Delayed evaluation via `exec` locations.
- new operator: concat ("verb—::—") both as left- and right-value
- new operator: subfield "verb—¡from...to¿—" both as left and right-value

Changes between version 1.2 and version 1.0 (Version 1.1 was internal):

- new unary operator "`-`" (needed for IEEE floating point since $-0 \neq 0 - 0$)
- changed "`^`" to "`**`"
- new binary operators "`^`" (exclusive-or) and "`^^`" (logical exclusive-or)
- numerous typo fixes

9 Appendix: Grammar of `nML`

What follows is a kind of typed EBNF grammar. An annotation like foo_{bar} means that "*foo*" is of type "*bar*". $(X)^*$ means "a sequence of 0 or more Xs". $X \mid Y$ means "either X or Y". $[X]$ means "one X or nothing".

machine-description $\Rightarrow$
  (*memory-spec*
  | *type-spec*
  | *mode-spec*
  | *op-rule*
  | *let-def*
  | *macro-def* $)^*$

memory-spec $\Rightarrow$
  `mem` $\mathit{name}_{mem\ type}$ `[` expr_{card} `,` name_{type} `]` $(\mathit{mem\text{-}attribute})^*$

mem-attribute $\Rightarrow$
  `volatile =` expr_{string}
  | `alias =` *location*

type-spec $\Rightarrow$
  `type` $\mathit{name}_{typespec}$ `=` $\mathit{expr}_{typespec}$

$\mathit{expr}_{typespec}$ $\Rightarrow$
  `bool`
  | `int(`expr_{card}`)`
  | `card(`expr_{card}`)`
  | `fix(`expr_{card}`,`expr_{card}`)`
  | `float(`expr_{card}`,`expr_{card}`)`
  | `[`expr_{int} `..` expr_{int}`]`
  | `enum(`$i_{card} \ldots i_{card}$`)`

expr_{type} $\Rightarrow$
  const_{type}
  | $\mathit{location}_{type}$
  | $\mathit{function\text{-}application}_{type}$
  | `if` expr_{bool} `then` expr_{type} `else` expr_{type} `endif`
  | *param-name* . *attrib-name*
  | `switch(`$\mathit{expr}_{select-type}$`)`
    {
      $(($`case` $\mathit{const}_{select-type}$ | `default`) : $\mathit{expr}_{type})^*$
    }

$\mathit{function\text{-}application}_{type}$ $\Rightarrow$
  $\mathit{name}_{type_1,\ldots,type_n \rightarrow\ type}$ `(`expr_{type_1} `, ... ,` expr_{type_n}`)`
  | expr_{type_1} $\mathit{name}_{type_1,type_2 \rightarrow type}$ expr_{type_2}

$location_{type} \Rightarrow$
    $name_{mem\ type}$ `[`$expr_{card}$`]`
    $\mid name_{mem\ type}$ [8]

$name_{int,int \rightarrow\ int} \Rightarrow$
    `+` $\mid$ `-` $\mid$ `*` $\mid$ `div` $\mid$ `mod`

$name_{card,card \rightarrow\ card} \Rightarrow$
    `&` $\mid$ `|` $\mid$ `^` $\mid$ `>>` $\mid$ `<<` $\mid$ `**`

$name_{bool,bool \rightarrow\ bool} \Rightarrow$
    `&&` $\mid$ `||`

$name_{bool \rightarrow\ bool} \Rightarrow$
    `not`

$name_{type,type \rightarrow\ bool} \Rightarrow$
    `>=` $\mid$ `>` $\mid$ `<=` $\mid$ `<` $\mid$ `==` $\mid$ `!=`

$name_{format-string,type,...,type \rightarrow\ string} \Rightarrow$
    `format`

$name_{type-specifier,type \rightarrow\ specified\ type} \Rightarrow$
    `coerce`

op-rule $\Rightarrow$
    and-rule
    $\mid$ *or-rule*

or-rule $\Rightarrow$
    `op` $name_{op}$ `=` id_{op} `|` ... `|` id_{op}

and-rule $\Rightarrow$
    `op` $name_{op}$ `(` id_{param} `:` $name_{typespec}$ `,` ... `,` id_{param} `:` $name_{typespec}$ `)`[9]
        $(attribute\text{-}def)^*$

mode-spec $\Rightarrow$
    mode-and-rule
    $\mid$ *mode-or-rule*

mode-or-rule $\Rightarrow$
    `mode` $name_{mode}$ `=` id_{mode} `|` ... `|` id_{mode}

[8]This abbreviation ($x \equiv x[0]$) is only valid if x denotes a memory base of size 1.
[9]The parameter list may have length 0.

mode-and-rule $\Rightarrow$

$\texttt{mode}\ name_{mode}\ \texttt{(}\ id_{param}\ \texttt{:}\ name_{typespec}\ \texttt{,...,}\ id_{param}\ \texttt{:}\ name_{typespec}\ \texttt{)}\ [\texttt{=}expr]$

$(attribute\text{-}def)^*$

attribute-def $\Rightarrow$

$name_{attr}\ \texttt{=}\ (expr \mid sequence)$

sequence $\Rightarrow$

$(statement\texttt{;})^*$

statement $\Rightarrow$

$location_{type_1}\texttt{=}expr_{type_2}$

$\mid \texttt{if}\ expr_{bool}\ \texttt{then}\ sequence\ [\texttt{else}\ sequence]\ \texttt{endif}$

$\mid \texttt{switch(}\ expr_{select-type}\ \texttt{)}$

$\{$

$((\texttt{case}\ const_{select-type} \mid \texttt{default})\ \texttt{:}\ actions)^*$

$\}$

let-def $\Rightarrow$

$\texttt{let}\ name_{type}\texttt{=}expr_{type}$

macro-def $\Rightarrow$

$\texttt{macro}\ name_{macro}\texttt{(}\ param\ \texttt{,...,}\ param\texttt{)}\ \texttt{=}\ expr$

10 Appendix: Selected Problems

This appendix shall show how concepts like data pipelining and interrupts, which are not directly supported, can be modelled by adding constant preambles and postludes *to each instruction*, i.e. by giving the root of the instruction tree an extra preamble (and/or epilogue) that contains a description of the 'additional machinery' that 'runs parallel' to the machine.

10.1 Pipelines

Consider a machine with a 3-cycle multiplication. How may this be modelled? Lets assume a 16 bit * 16 bit multiplication with a 32-bit result in a register pair. This is written down as

```
   mult D1,D2,D3
```

and has the semantics "compute D1*D2 and put the result into the register-pair D3/D4". The storing of the result shall occur 3 cycles later, i.e. in the program

```
   move #1,D3
   move #2,D4
   mult D1,D2,D3       \ start multiplication...
   add  #1,D3          \
   move D3,D1          \ D1=1
   move D4,D2          \ D2=2
   \ now the multiplication results are transferred to D3,D4
   move D3,D0          \ move the high word of D1*D2 to D0
```

So two tasks have to be modelled: first the computing over a time of 3 cycles, and then the storing. Assuming that the multiplication is fully pipelined, one could write something like

```
mem D[16,word]              \ a few registers

mem mult_dst0[1,card(4)]
mem mult_dst1[1,card(4)]
mem mult_dst2[1,card(4)]
mem mult_dst3[1,card(4)]

mem mult_flag0[1,bool]
mem mult_flag1[1,bool]
mem mult_flag2[1,bool]
mem mult_flag3[1,bool]

mem mult_output0[1,long]
mem mult_output1[1,long]
mem mult_output2[1,long]
mem mult_output3[1,long]
```

```
op instruction(i:rest_instruction) \ root of the instruction tree
action={
        mult_flag0=0;              \ no multiplication started
        i.action;
        if mult_flag3 then
          D[mult_dst3]=mult_output3 >>16;       \ high word
          D[mult_dst3+1]=mult_output3 & 0xffff; \ low word
        endif;
        mult_output3= mult_output2; \ advance pipeline
        mult_dst3   = mult_dst2;
        mult_flag3  = mult_flag2;
        mult_output2= mult_output1;
        mult_dst2   = mult_dst1;
        mult_flag2  = mult_flag1;
        mult_output1= mult_output0;
        mult_dst1   = mult_dst0;
        mult_flag1  = mult_flag0;
       }

op rest_instruction = ... | mult | ...  \ many different operations,
                                        \ amongst them mult
op mult(x:card(4),y:card(4),dst:card(3))
action={
        mult_output0=D[x]*D[y];
        mult_dst0=dst;
        mult_flag0=true;
       }
```

How does it work? The pipeline is modelled by 3 sets of registers: one modelling the data part of the pipeline, the second one remembers the destination register, the third one flags whether a multiplication is going on at all. The 'mult' operation multiplies the contents of the registers, inserts the result into the first step of the pipeline, stores the destination register, and sets the flag. Now, at the beginning of *each* instruction, the multiplication pipeline is advanced one step (regardless of whether it is filled or not). So, the cycles go as follows:

```
   mult D1,D2,D3  mult_output0=D1*D2;
                  mult_output1=mult_output0
-------------------------------------------
   add  #1,D3
                  mult_output2=mult_output1
-------------------------------------------
   move D3,D1
                  mult_output3=mult_output2
-------------------------------------------
   move D4,D2
                  D3,D4=mult_output3
-------------------------------------------
   move D3,D0
```

How does one model a not-pipelined multi-cycle operation? One way is to use a counter instead of a pipeline structure:

```
mem D[16,word]

mem mult_reg[1,long]
mem mult_cnt[1,card(3)]
mem mult_dst[1,card(4)]

op instruction(i:rest_instruction)
action={
        i.action;
        if mult_cnt==1 then
          D[mult_dst]=mult_reg >> 16; \ high word
          D[mult_dst]=mult_reg & 0xffff; \ low word
        endif;
        if mult_cnt>0
        then mult_cnt=mult_cnt-1
        endif;
       }

op rest_instruction = ... | mult | ...

op mult(x:card(4),y:card(4),dst:card(3))
action={
        mult_cnt=4;
        mult_reg=x*y;
        mult_dst=dst;
       }
```

Here, a mult_cnt of 0 marks an inactive pipeline. If the pipeline is active and the count is on 1, the value is stored.

10.2 Interrupts

One can model interrupts in about the same way as pipelines. Assume an interrupt register that may hold a value of 0 or an interrupt number that serves as index into some vector array stored at address 256.

```
mem interrupt_register[1,card(4)] volatile="irq"

op instruction(i:rest_instruction)
action={
        i.action;
        if interrupt_register!=0
        then STORED_PC=PC;
             PC=M[interrupt_register<<2+0x100];
```

```
          interrupt_register=0;
      endif;
     }
```

The interrupt-register is marked as "volatile", i.e. "changing its value". If some non-0 value appears, the PC is stored in some intermediate location (or put on the stack or whatever) and changed to the address found at the index. Of course, on a real machine much more happens: the current CPU state is stored, special mode bits are set, interrupts may be masked, etc.

Index

References

[1] Markus Freericks:
The nML Machine Description Formalism,
Forschungsberichte des Fachbereichs Informatik Nr.91-15

[2] Markus Freericks:
The nML Machine Description Formalism (updated Version 1.2),
in: ESPRIT-II Project 2260 SPRITE Progress Report (Incl. Addendum 2 to T.A. 3) for Period June 1992-November 1992, Report No. PR-4.2, 1. Dec. 1992, Author: SPRITE Consortium, Editor: Patrick Pype (Project Manager)

[3] A. Fauth, M. Freericks, A. Knoll:
Generation of Hardware Machine Models from Instruction Set Descriptions,
in: VLSI Signal Processing, VI, Eggermont et.al. (eds), IEEE Signal Processing Society, 1993

[4] Peter J. Ashenden:
The VHDL Cookbook
First Edition, July 1990
Dept. Computer Science, Univ. of Adelaide, South Australia

[5] CAD Language Systems Inc:
VHDL Language Reference Manual
Draft Standard 1076/A, 31 December 1986

[6] R.G.G. Cattell:
Automatic Derivation of Code Generators from Machine Descriptions, in: ACM Transactions on Programming Languages and Systems 2(2), April 1980, pp. 173-190

[7] J.W. Davidson and C.W. Fraser:
The Design and Application of a Retargetable Peephole Optimizer, in: ACM Transactions on Programming Languages and Systems 2(2), April 1980, pp. 191-202

[8] Edif Steering Committee:
EDIF Specification Version 1.1.1
June 1986

[9] Edif Steering Committee:
EDIF Electronic Design Interchange Format Version 2.0.0 Draft
December 1986

[10] Gilberto File:
Theory of Attribute Grammars
Ph.D. thesis, Technische Hogeschool Twente, 1983

[11] D. Knuth:
(Knuths attribute grammar paper. CACM anno '65, it was, i think.)

[12] Robert Severyns, Eric Willems:
HILARICS-2: The Language
December 7, 1989, Preliminary Release V1.0

[13] Richard M. Stallman:
Using and Porting GNU CC
version tagged as “last updated 12 September 1989, for version 1.36”