Describing Instruction Set Processors Using nML

A. Fauth[1] J. Van Praet[2] M. Freericks[1]

[1]Institut für Technische Informatik
Tech. Univ. Berlin, Franklinstr. 28/29
D–10587 Berlin, Germany

[2]IMEC
Kapeldreef 75
B–3001 Leuven, Belgium

Abstract

Programmable processors offer a high degree of flexibility and are therefore increasingly being used in embedded systems. We introduce the formalism nML *which is especially suited to describe such processors in terms of their instruction set, an* nML *description is directly related to the standard description as found in the usual programmer's manuals. The* nML *formalism is based on a mixed structural and behavioural model facilitating exact yet concise descriptions. The philosophy of* nML *is already applied in two approaches to retargetable code generation and instruction set simulation.*

1 Introduction

In consumer electronics and telecommunications high product volumes are increasingly combined with short life-times and high system complexity. The pressure on development times together with the demand to react on late specification changes make mask or field programmability a desired feature. The thereby obtained flexibility not only helps to shorten the design cycle, but also allows for the reuse of carefully optimised hardware designs. The different kinds of programmable DSP core processors frequently used include application specific instruction set processors (ASIPs) [23] and commodity digital signal processors [19, 20]. ASIPs are a blend of full-custom circuits and "off-the-shelf" DSPs. They offer instruction sets which are (together with chip area and power dissipation) especially optimised for a small number of applications. Commodity DSPs are mostly fixed designs that allow for "customer" designed add-ons. In hardware/software co-designed ICs, critical parts of a system featuring such a programmable processor are often implemented in hardware. This is done by adding dedicated hardware in the form of custom accelerator data paths, making the design a heterogeneous IC architecture [14].

Code generators, instruction set simulators and assemblers are the key tools that aid the designer when developing the software. However, when the hardware is customised conventional compilers and simulators are deficient, while coding complex applications in assembly language by hand is error-prone and very costly. The only acceptable solution for mapping a small number of algorithms onto such a processor is to have a set of *retargetable* tools. Then, such tools can even aid the evaluation of different hardware variations by providing quality measures in terms of code size, execution time and resource utilisation.

To minimise the retargeting effort, the target machine description used by the tools should be concise yet powerful enough to contain all information necessary for efficient code generation and reliable instruction set simulation. We believe that the machine is best described by giving the details of the instruction set together with a high-level structural model of the processor's architecture. This contains all information at the level typically available in programmer's manuals. These observations have led us to develop the machine description formalism nML.

We will first describe some related work, then present the basic philosophy of nML and end with an overview of applications of nML.

2 Related work

A lot of work has been done on processor description languages. In the sequel we will discuss a number of approaches that are relevant in the context of our work.

ISP [5] is based on the assumption that a programmable instruction set processor can be described by specifying the operations that can be performed together with the rules of decoding. A procedural description of the interpretation and execution of the instructions is thus given. *ISPS* [3] is a descendant of ISP with additional features to support "local units". These units can either be sequentially executed factorisations of a certain behaviour (called *procedures*) or units that operate concurrently to the "main circuit" (called *processes*). *MIMOLA* [22, 4] describes the target processor in terms of a netlist consisting of a set of modules and a detailed interconnection scheme.

The decoding logic is also explicitly described. *VHDL* is a standardised language with considerable semantic richness. Because VHDL can be used at different levels, several description styles have evolved. However, if different tools support different styles or language subsets, the benefits of the standardisation vanish. All of the abovementioned hardware description languages require a detailed knowledge of the processor netlist and the instruction decoder. However, this information is usually not available nor is it necessary for the tasks of code generation and instruction set simulation.

In traditional software compilers (for CISC architectures), string and tree grammars have found to be an adequate notation for formalising the mapping of machine instructions to application programs [13, 12, 1]. Hence their use is limited to this single phase of a code generator called *code selection*. *Register allocation* and *instruction ordering* usually need knowledge about other concepts of the target machine and require different descriptions.

3 The nML Formalism

The information typically available in a programmer's manual consists of a list of instructions and the corresponding register transfers, binary coding and assembly language mnemonics. Often, also a programmer's model of the machine is provided, usually in terms of a coarse schematic showing the registers, the functional units and their basic interconnection scheme. Concerning pipelining, only the possible *hazards* [15] are highlighted (e.g. branch penalties and specific timing of read/write cycles for address registers). Other unique features such as zero-overhead loops and conditional computations are also described from a programmer's view point. However, a *detailed* description of the datapath is usually not available, nor is a description of the controller or microsequencing logic. Nevertheless, the information at hand is sufficient for code generation and instruction set simulation.

Using nML, hardware structure is described together with execution behaviour and the coding of all instructions: nML is based on a *mixed behavioural/structural* paradigm.

On one hand, a *skeleton* of the target machine *structure* is constructed by declaring all storage entities. On the other hand, register transfers between these storage entities describe the exact *execution behaviour* of the machine including side-effects (such as the setting of condition codes), the complete set of addressing modes, the possibilities of controlling the program flow and the encoding of the instructions.[1]

[1] Note that nML allows for bit-true modelling of the target processor, which is of high importance especially in the field of DSP.

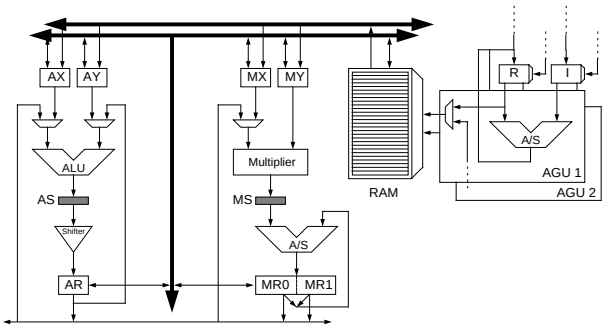

Figure 1: The example processor.

3.1 Breaking Down the Instruction Set

When describing an instruction set, a top-down approach is advocated. In nML, a hierarchical structure can be imposed onto the instruction set description. Consequently, the description is partitioned into *rules*. There are two kinds of rules,

- *OR-rules* which list all *alternatives* for an instruction part, and
- *AND-rules* which describe the *composition* of instruction parts.

These rules form a grammar from which each possible derivation represents one *legal* instruction. It will be shown that the structure of the grammar resembles the structure of the instruction set.

Figure 1 shows an example processor which basically consists of a controller, two (pipelined) datapaths, two address generation units and a centralised RAM. However, the instruction set poses severe limitations on the set of combinations of operations that can be executed in parallel. These limitations are called *encoding restrictions*. Figure 2 comprises the possible instruction formats (separated by horizontal lines). The instruction word consists of 18 bits (where vertical lines separate independent fields).

An nML description is typically constructed by analysing the instruction set of the target machine proceeding top-down. For our example, the aforementioned encoding restrictions are easily reflected in the nML description by capturing the top-level classification in an OR-rule:

```
opn instruction = computemove | moveabs | ctrl
```

Either a computation with a data move in parallel, a sole data move with absolute addressing or a control-related instruction can be executed. Further zooming into the first category of instructions, two *orthogonal* parts are found, i.e. two parts of the instruction that can be controlled independently. This is described in an AND-rule:

```
opn computemove(c:compute,m:move)
```

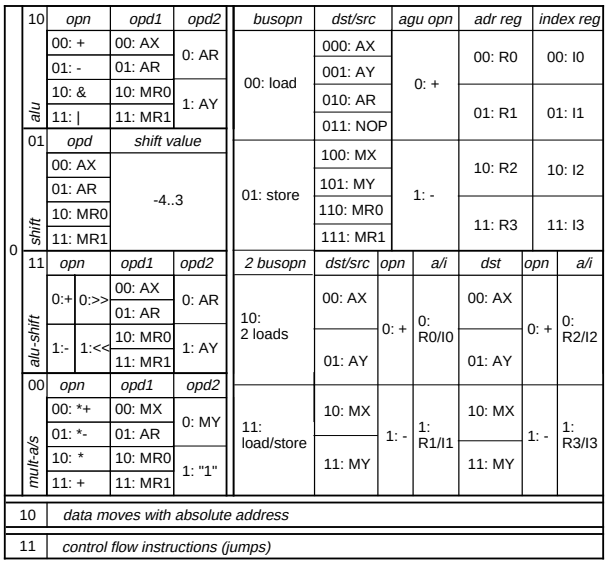

Figure 2: The instruction set table.

The parts of the instruction that are composed with this rule are listed in its *parameter list*. The declaration of a parameter consists of an *instantiation name* and a reference to some other rule or to a data type (in case of an *immediate* operand). The description of each orthogonal part is encapsulated in a rule of its own. It often occurs that such a part is even referred to by more than one rule. In our example, the operand class {`AX`, `AR`, `MR0`, `MR1`} is subsumed in an addressing mode rule:[2]

```
mode lopd = AX | AR | MR0 | MR1
```

There are three references to this rule. One reference for *opd*1 of *alu*, one for *opd* of *shift* and one for *opd*1 of *alu-shift*.

As a consequence of the hierarchy and the factorisations, the descriptions are concise and easily maintainable.

The names and parameters of rules themselves have no meaning besides structuring the description; all semantics are held in attributes attached to the rules. Three attributes are pre-defined: `action` to specify the execution behaviour, `image` to define the binary coding and `syntax` to describe the assembly language mnemonics.

3.2 Specification of Execution Behaviour

For nML, a basic model of execution is presupposed: a machine executes a program consisting of a *single thread of instructions*. These instructions are stored in a memory from which they are fetched using a *program counter* (PC). Hence the program flow can be changed by writing to the PC. Once an instruction is

[2] This rule is represented by the highlighted node in figure 3.

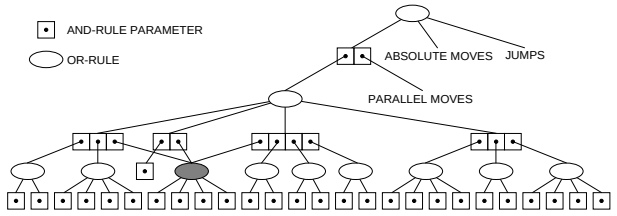

Figure 3: Structure of the instruction set description.

fetched from memory, it is decoded and the the appropriate *register transfers* are executed. The execution of these register transfers completely determines the *behaviour* of the machine. The *effects* of the execution are entirely captured in the contents of the storages at the end of a machine cycle. Thus the storages represent the state of the machine and an instruction can be seen as a state *transition function*. This information is stored in the `action` attribute. Its value consists of a sequence of statements.[3] A statement must be either an *assignment* or a *conditional* as shown in the following two rules.[4]

```
opn shift(o:lopd,sh:int(3))
  action={
     AS = o; AR = AS << sh;
  }
opn alu(o1:lopd,o2:ropd,op:card(2))
  action={
    switch op             // depending on op...
    case 0: AS = o1 + o2; // ...a different...
    case 1: AS = o1 - o2; // ...operation...
    case 2: AS = o1 & o2; // ...is executed...
    case 3: AS = o1 | o2; // ...in the alu.
    end;
    AR = AS;
  }
```

The predefined *operators* to form expressions include the common "C" operators plus some DSP-related extensions (e.g. exponentiation, bit rotation, bit string selection and concatenation).[5]

The definition of an attribute can include references to attributes defined by the parameters of the rule.

```
opn computemove(c:compute,m:move)
  action={ c.action; m.action; }
```

The above defines the `action` of `computemove` as the sequence of the `action`s of the instances of `compute` and `move`.

The binary coding and the assembly language mnemonic are captured in the `image` resp. the `syntax` attribute. The value of the `image` attribute is a bit string (or a cardinal number) and the value of the

[3] In nML, maximum possible parallelism is implicit.

[4] There is also an `if-then-else` clause.

[5] Other operations (e.g. for the descriptions of accelerator paths) can be added as *canonicals*, i.e. operations without predefined semantics.

`syntax` attribute is a string.

```
opn computemove(c:compute,m:move)
  image="0"::c.image::m.image
  syntax=format("%s || %s",c.syntax,m.syntax)
```

The `image` attribute attached to this rule expresses that the binary code for this instruction part consists of a fixed single bit prefix "0" concatenated with the `image` attributes of the two components. The definition of the `syntax` attribute is similar. (The function `format` resembles the "C" library function `printf`).

In addition to the aforementioned `opn` rules, there are rules to support the description of addressing modes. These `mode` rules behave similar to `opn` rules but have an additional default attribute which contains an *effective address* expression. The parallel data moves of our example processor use indirect addressing to compute the effective address.

```
mode indinc(j:card(2),k:card(2)) = m[r[j]]
  action={ r[j]=r[j]+i[k]; }
  image=format("0%b%b",j,k)
  syntax=format("(R%d++I%d)",j,k)
```

This rule specifies the effective address as well as some code that is used to *update* the address register. It is used in contexts such as:

```
mode adrmode = indinc | inddec

opn load(r:regm,a:adrmode)
  action={ r=a; a.action; }
  image=format("00%b%b",r.image,a.image)
```

3.3 Specification of Structure and Timing

Storages are structural entities and are used to establish a skeleton of the over-all structure of the machine. The behaviour as introduced in the previous section is basically described by register transfers between these storages.[6]

A storage is declared by giving a name, the size and the element type.

```
mem m[1024,int(16)]
reg r[4,fix(1,31)]
```

This defines a memory `m` of 1K with elements of 16 bit integer numbers and a four element register file `r` of 32 bit fixed-point numbers.

In nML, the duration of an operation's execution is assumed to be zero. The timing is modelled by specifying a *delay* for each storage.

The storages described so far are *static*, i.e. they hold their value until explicitely overwritten. To model pipelines and similar concepts in nML, *transitory* storages [16] are introduced. These storages hold values only for a fixed time, i.e. once a value is written into a transitory and a specific number of cycles have passed, the value is available only during a single cycle. Then, their contents become undefined. By specifying a delay of zero for a transitory, a wire or a bus is described. In order to model simultaneous execution of operations, the designer can specify that the outputs of several transitories are synchronized. Such groups of transitories divide a pipelined datapath into *pipe stages* (see figure 4).

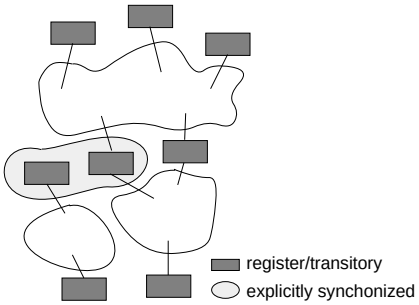

Figure 4: Pipe stages.

For our example, we would describe the following two transitories

```
trn as[1,int(16)] delay=1
trn ms[1,int(32)] delay=1 sync=as
```

The two transitory storages `as` and `ms` are declared with a delay of one. Both are synchronized dividing the datapath into two pipe stages.

Such descriptions fit naturally in the style of nML since they combine the basic execution model (i.e., storages represent the state of the machine) with the basic structural model (i.e., storages establish a structural skeleton). All necessary information about timing and resource usage can be easily extracted from such a mixed structural/behavioural description.

4 Applications of nML

The formalism nML is currently used at IMEC and at TU Berlin. This section explains how it is used in the contexts of retargetable code generation and instruction set simulation.

The group at TUB has developed the retargetable code generator CBC [6, 8] and the retargetable instruction set simulator SIGH/SIM [21]. IMEC has developed the retargetable code generator CHESS [24, 18] and plans to implement an instruction set simulator in the near future.

The specification of an instruction's action is used both for the mapping required in the code generator and the simulation of the instruction in an appropriate environment. For both applications appropriate models are extracted automatically from a *single* machine description [7, 24]. Note that the use of nML does not presuppose a certain compiler model and that specific aspects needed by different tools can be easily

[6] By allocating an appropriate set of functional units for each set of register transfers between the same storages a netlist can be constructed [7]. Hence nML can also be used for processor implementation.

extracted from a single description thereby guaranteeing consistency between models.

A main intention of work related to nML is to have clear semantics defined. It is believed that this is essential to *formally* correct code generation and reliable instruction set simulation [11]. The basic difference in the use of nML in the two applications lies in the interpretation of the action attribute. The CHESS compiler uses a centralised library [17] on which all tools rely. The semantics of all operations of the target processor are described in terms of operations listed in this library. In contrast, the semantics for use in CBC are given in a "language report" [10].

A description of the ADSP-2100 – a DSP from Analog Devices [2] – is approximately 900 lines of which 300 lines are comments. To describe the complete instruction set 59 rules are necessary out of which 44 are operation definitions (`opn`) and 15 addressing mode definition (`mode`). The time to develop the description was less than one month.

5 Summary

The machine description formalism nML has been described along with its design philosophy. Situated at the level of information found in programmer's manuals, nML allows for concise specifications of programmable instruction set processors. The analysis of the instruction formats leads straightforwardly to the organisation of the description. The mixed structural/behavioural approach taken supports flexible modelling of even irregular processor architectures. Current research is concerning the addition of *inherited* attributes to nML to facilitate the specification of repetitive structure [9].

Acknowledgment

This research has been sponsored by the E.U. through the ESPRIT 2260 project ("SPRITE") and the Large Installations Plan. The first author would also like to thank the Belgian breweries and "frituren" for supporting his stay in Belgium.

References

[1] A.V. Aho, M. Ganapathi, S.W. Tjiang, "Code Generation Using Tree Matching and Dynamic Programming", ACM TOPLAS, Vol. 11, No. 4, October 1989, pp. 491-516

[2] Analog Devices, "ADSP-2100 User's Manual", 1989

[3] M.R. Barbacci, "Instruction Set Processor Specifications (ISPS): The Notation and Its Applications", IEEE Transactions on Computers, Vol. C-30, No. 1, January 1981, pp. 24-40

[4] S. Bashford et al., "The MIMOLA Language Version 4.1", Tech. Rep., Lehrstuhl Informatik XII, Univ. Dortmund, September 1994

[5] C.G. Bell, A. Newell, "Computer Structures: Readings and Examples", MacGraw-Hill, 1971

[6] A. Fauth, A. Knoll, "Automated generation of DSP program development tools", in Proc. IEEE ICASSP-93, May 1993

[7] A. Fauth, M. Freericks, A. Knoll, "Generation of Hardware Machine Models from Instruction Set Descriptions", VLSI Signal Processing VI, 1993, pp. 254-250

[8] A. Fauth, G. Hommel, C. Müller, A. Knoll, "Global Code Selection for Directed Acyclic Graphs", in Proc. Compiler Construction 94, Edinburgh, Scotland, LNCS 786, Springer, pp. 128-142

[9] A. Fauth, J. Van Praet, M. Freericks, "Describing Instruction Sets Using nML (Extended Version)", Tech. Rep., TU Berlin, Fachbereich Informatik, Berlin, also available from IMEC, Leuven, Belgium

[10] M. Freericks, "The nML Machine Description Formalism", Tech. Rep. 1991/15, TU Berlin, Fachbereich Informatik, Berlin, 1991

[11] M. Freericks, A. Fauth, A. Knoll, "A Basic Semantics for Computer Arithmetic", Techn. Rep.(under preparation), Univ. Bielefeld, Fakultät für Technische Informatik

[12] M. Ganapathi, C.N. Fischer, J.L. Hennessy, "Retargetable Compiler Code Generation", Computing Surveys, Vol. 14, No. 4, December 1982, pp.573-592

[13] R.S. Glanville, S.L. Graham, "A new method for compiler code generation (Extended Abstract)", in Proc. POPL 78, 1978, pp. 231-240

[14] G. Goossens, I. Bolsens, B. Lin, F. Catthoor, "Design of heterogeneous ICs for mobile and personal communication systems", in Proc. IEEE ICCAD-94, November 94, pp. 524-531

[15] J.L. Hennessey, D.A. Patterson, "Computer architecture: a quantative approach", Morgan Kaufmann Publ., 1990

[16] D. Landskov, S. Davidson, B. Shriver, P. Mallett, "Local microcode compaction techniques", ACM Computing Surveys 12(3), Sept. 1980, pp. 261-294

[17] D. Lanneer, Design Models and Data-Path Mapping for Signal Processing Architectures, Ph.D. thesis, K.U. Leuven-IMEC, March 1993

[18] D. Lanneer et al., "Data routing; a paradigm for efficient data path synthesis and code generation", in Proc. 7th Int. Symp. on High-Level Synth., May 1994, pp. 17-22

[19] E.A. Lee, "Programmable DSP architectures: Part I & II", IEEE ASSP Magazine December 1988, pp. 4-19, and January 1989, pp. 4-14

[20] EDN, "EDN's 1994 DSP chip directory", compiled and edited by J.P. Leonard, June 1994, pp. 75-135

[21] F. Löhr, A. Fauth, M. Freericks, "SIGH/SIM – an Environment for Retargetable Instruction Set Simulation", Tech. Rep. 1993/43, TU Berlin, Fachbereich Informatik, Berlin, 1993

[22] P. Marwedel, "Tree-Based Mapping of Algorithms to Predefined Structures", in ICCAD-93, pp. 586-593

[23] P.G. Paulin et al., "DSP Design Rool Requirements for Embedded Systems: a Telecommunications Industrial Perspective", J. VLSI Signal Proc. (Special Issue on Synthesis for Real-Time DSP), Vol. 9, No. 1, 1995

[24] J. Van Praet et al., "Instruction Set Definition and Instruction Selection for ASIPs", in Proc. 7th Int. Symp. on High-Level Synth., May 1994, pp. 11-16