Generation of Hardware Machine Models from Instruction Set Descriptions

A. Fauth, M. Freericks, A. Knoll

This paper describes how a modular machine description, which specifies the functionality and the binary representation of an instruction set, can be transformed into a hardware model. This model is built from few generic hardware entities (registers, memories, arithmetic/logic operators, selectors and connections) and may eventually serve as an input to high-level hardware synthesis tools. The transformation steps on the way from the machine description to the hardware model are explained by giving an example.

Introduction

In recent years much work in hardware design tools for digital signal processing (DSP) was spent on high-level synthesis: on the basis of a behavioural description a piece of hardware is synthesized that exactly meets the requirements of the description. In other words, a circuit is generated which is capable of performing a fixed function. Although this circuit may be close to the optimum solution in terms of processing speed, it cannot be reused for similar applications. One way out of this predicament is the design of instruction set programmable processors targetted at a rather narrow domain of applications but not limited to one specific application. When designing such *domain specific processors*, parts of the required capabilites may be realized by programmable circuitry, other parts by fixed function blocks. To achieve rapid design cycles, the key issue with this approach is the definition of a compact and readable architecture description language.

The `nML` machine description formalism [1, 2] describes a processor architecture in terms of its instruction set (IS). The IS is represented by an attributed grammar with the possible derivations reflecting the set of legal instructions and the attributes determining the execution semantics and the binary encoding of each instruction. By providing a structured description of the IS (instead of a flat enumeration of all instructions), the designer can express sharing of common properties between instructions. The descriptions are thus shorter, easier to understand and to modify.

VLSI synthesis tools such as CATHEDRAL-2ND [3] and the MIMOLA system [4] as well as retargetable compilers for fixed DSP cores, such as the CBC system [2, 5], are based on detailed hardware descriptions. A major statement of our design philosophy is that the structure of the hardware is neither directly given by the description of "modules" (as in MIMOLA) nor by the structure of the IS description. The only "hints" to the structure are inside the encoding of the instructions, i.e. the implicit definitions of (global and local) controllers. This paper describes the transformation of an `nML` machine description into a model of the hardware. The different phases of the process are explained and illustrated in terms of graphical representations of the hardware entities.

Hardware Model Entities

A hardware model is composed of individual entities (HMEs), which are connected via signals. For most architectures, the following generic HMEs suffice (see figure 1):

- Memory banks with a fixed width (type), size and sets of read/write ports.
- Basic arithmetic/logic elements (add, and, not, shift, etc.) and unspecified ("canonical") blocks.[1]
- Signals connecting hardware blocks. A signal has one source and can have an arbitrary number of destinations.
- Sequencing edges used to connect arbitrary elements of the machine preserving synchronized read and write accesses. A sequencing edge leading from an HME A to a block B forces the execution of A to take place prior to the execution of B. The presence of a signal (i.e. a data-flow edge) implies a sequencing edge, because a signal cannot be consumed before having been produced.[2]
- Conditional scopes, i.e. demultiplexer/multiplexer pairs that contain a number of sets of HMEs each and are controlled by a signal that selects one of the sets. Signals and sequencing edges can enter and leave such scopes.

[1] These can be used to describe the operations that are executed on accelerator paths.

[2] Alternatively, a sequencing edge can be interpreted as a degenerated signal, i.e. one that contains no information besides the sequencing that is implied by its very existence.

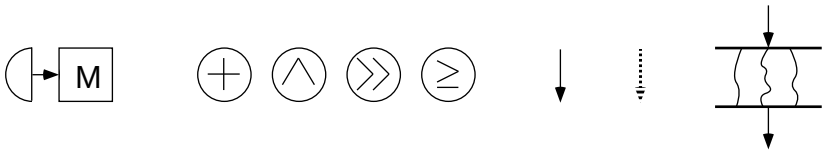

Figure 1. HMEs for a memory, four different ALU components, a signal, a sequencing edge and a conditional scope.

The nML Description Formalism

An **nML** [1, 2] description consists of a list of memory location bases (i.e. sets of memory units addressable under a common name) and an instruction tree.[3] This tree consists of and-/or-rules and attributes attached to the rule definitions. The alternatives of an or-rule propagate all attributes to the referencing rule. Each machine instruction has a corresponding instantiation of the rule tree. The instruction's binary encoding and semantics are defined by the `image` and `action` attributes that can be computed for this instance. Our final machine model will contain an *instruction register* that holds the "current" instruction code and *decoding logic* that controls the different conditional contexts and ALU units. By introducing this logic at an early stage of the transformation trajectory, the two attributes `image` and `action` are modeled in a unified internal representation of the architecture throughout the whole process. Previous approaches of deriving architecture models [6][7] analyze the attributes as separate entities and are usually organized around the treatment of one attribute (`action` or `image`) only.

Throughout this paper, the transformation of the following example will be shown. The machine, though small, represents all major problems encountered in real-life architectures.[4] It can execute one data transfer or one ALU action or one conditional jump per instruction.[5]

```
\\ type declarations
type word=card(16)                  \\ basic data type
type absa=card(9)                   \\ absolute memory addresses
type disp=int(4)                    \\ displacement
type off=int(6)                     \\ relative jump addresses
```

[3] This tree can also be understood as a directed acyclic graph (DAG) due to the sharing of properties. We prefer the notion of a tree because all shared properties are actually instantiated at every distinct use.

[4] Note that the **nML** description of the well known 6502 CPU fits on four pages.

[5] The only feature not covered in the example is parallelism, e.g. parallel data transfers and ALU operations. However, these pose no problem for our approach.

```
\\ memory definitions
mem PC[1,word]                      \\ the PC points to the "next instruction"
mem R[16,word]                      \\ the register file
mem M[65536,word]                   \\ the RAM

\\ latch definitions
mem L1[1,word]                      \\ 'left' ALU operand
mem L2[1,word]                      \\ 'right' ALU operand
mem L3[1,word]                      \\ ALU result

\\ instruction tree
op instruction = move | alu | jump  \\ the top node (root of the tree)
op move(lors:bool,r:reg,m:mem)      \\ addressing modes are referenced ...
  action={                          \\ ... via parameters
    if lors                         \\ a boolean determines ...
    then r=m;                       \\ ... whether a load ...
    else m=r;                       \\ ... or a store is performed
    end;
    m.update;}                      \\ update code for post-increment
  image=format("0%b%b%b",lors,r.image,m.image)
op alu(s1:reg,s2:reg,d:reg,a:aluop) \\ common to all ALU instructions
  action={
    L1=s1; L2=s2;                   \\ the operands are copied to latches
    a.action;                       \\ the 'aluop' action is performed here
    d=L3;}                          \\ the result is written back
  image=format("10%b%b%b%b",s1.image,s2.image,d.image,a.image)
op jump(s1:reg,s2:reg,o:off)        \\ the jump instructions
  action={
    if s1>=s2                       \\ if compared with itself, always true
    then PC=PC+o;                   \\ jump relative to PC
    end;}
  image=format("11%b%b%b",s1.image,s2.image,o)
op aluop = and | add | sub | shift  \\ possible ALU operations
op and()       action={L3=L1&L2;}   image="00"
op add()       action={L3=L1+L2;}   image="01"
op sub()       action={L3=L1-L2;}   image="10"
op shift()     action={L3=L1<<L2;}  image="11"

\\ addressing mode definitions: effective address is value of the rule name
mode reg(i:card(4)) = R[i]          \\ register direct mode
  image=format("%b",i)
mode mem = ind | post | abs         \\ RAM access modes
mode ind(r:reg,d:disp) = M[r+d]     \\ indirect with displacement
  update={}                         \\ no update-code
  image =format("0%b%b0",r.image,d)
mode post(r:reg,d:disp)= M[r+d]     \\ post-increment with displacement
  update={r=r+1;}                   \\ index register is incremented
  image =format("0%b%b1",r.image,d)
mode abs(a:absa)= M[a]              \\ absolute addressing
  update={}                         \\ no update-code
  image =format("1%b",a)
```

Transformation Phases

The transformation consists of three distinct phases. In the first phase, the `nML` file is parsed and a direct internal representation is constructed. In the second phase, a tentative model of the hardware is constructed. This model is hierarchical: one hardware modeling cell (HMC) cell is generated for each `op`-/`mode`-rule. In the final stage, this model is flattened and refined; the total number of HMEs is reduced, e.g. by combining HMEs of similar behaviour, such as operators for addition and subtraction.

Establishing the Primary Data-Base: After parsing, the `nML` description is represented by a set of attribute definitions, node derivations and a symbol table. The parse-tree is analyzed to propagate information about the type and usage of attributes. Attributes are not predefined; not every rule node must contain them. Only the top node of the description tree (i.e. `instruction`) is required to have an `action` and an `image` attribute. Attributes are often defined in terms of auxiliary attributes.[6] Therefore, it is necessary to determine the final usage of each attribute: it can be part of the `image`, part of the `action`, or neither (i.e. unused). This is accomplished by a simple top-down tree traversal marking each attribute according to its usage.

Hierarchic Cell Construction: The second phase of the transformation process generates one hardware modeling cell (HMC) for every attribute that specifies parts of the `action`.[7] The HMEs within an HMC are connected by data-flow edges corresponding to the assignments and by sequencing edges corresponding to the statement sequencing (";") in the `nML` description. Hierarchy is introduced by referenced attributes (i.e. attributes of parameters to the rule). Each HMC has three interfaces: the *input* interface, the *output* interface and the *image* interface.

The input interface represents the "starting point" of the HMC. It is connected to the "first" HMEs. Likewise, the output interface is connected to the "last" HMEs. All connections between HMCs are routed via the input and output interfaces. Depending on whether the HMC represents an `action` or a `mode`-value, the connecting entities are sequencing edges or signals. The image interface is a set of control signal input ports that are connected to referenced HMCs (i.e. HMCs generated from `action` attributes of parame-

[6] In the example, the update code for the addressing modes is defined by an extra attribute `update` which is used inside `move.action`.

[7] An `op`- or `mode`-rule can thus be represented more than once in the generated set of HMCs, if it has more than one attribute that contributes to the `action` of `instruction`.

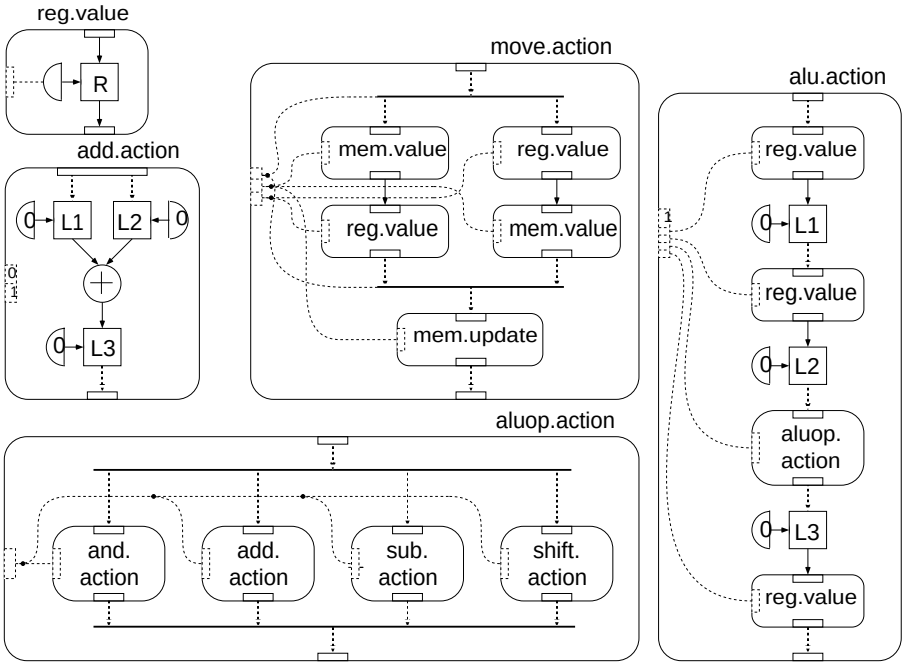

Figure 2. The `image` interfaces are represented by the ports at the left of each HMC feeding (dashed line) constant generators (drawn as half-circles), referenced HMCs or demultiplexer/multiplexer pairs.

ters to the rule) and to HMEs (then serving as constant generators). The image interface is used to model instruction word field conflicts. It allows a direct representation of encoding restrictions. Due to the hierarchy of an `nML` description, the `image` of every rule holds all information necesssary to specify the behaviour of the rule's `action`.

As a first example the HMC of the register direct addressing mode is considered (see `reg.value` in figure 2). This generic cell has signals connected to the input and output interfaces because it can be used in read and write contexts. Later, if only one of these lines is used, the other is replaced by a sequencing edge. The `add.action` HMC is exemplary for all `aluop`-rules. Sequencing edges lead from the start port to both input latches because no ordering relation can be deduced from the `nML` description. The read operations can thus be executed simultaneously. Eventually, most sequencing edges will be removed by introducing data dependencies. The HMC for `move.action`

shows the transformation of a conditional expression (`if`[8]). Control flows into a demultiplexer, enters one of the two possible paths, and merges at the multiplexer to flow through the `update` cell and leave the HMC. The bit that decides whether a load or a store is to be performed is extracted directly from the `image`. Both paths contain one instance of `mem.value` and `reg.value` each. At this point of the construction, no attempt is made to merge these copies. The HMC of the `alu.action` attribute shows the sequencing of the assignments and the reference to `aluop.action`. The different fields in the `image` attribute are connected to the corresponding embedded HMCs. The constant `"1"` in the interface is used in the `instruction` rule to distinguish `alu` instructions from `jump` and `move` instructions. The `aluop.action` is constructed out of an or-rule. This also results in a demultiplexer/multiplexer pair. Control flow enters the HMC and is directed to one of the embedded HMCs representing the actual ALU actions. The `image` is transferred to all embedded HMCs, because they all have the same `image` interface. The decision which of the four actions will be performed is done by a *controller*. The key insight into constructing the controller is the fact that the `image`s of the `action`s contain distinct manifest values. One of these is the `"01"` in the `image` of `add.action`. By examining the `image`s of the HMCs, these bits can be extracted from their `image` ports and used to control the demultiplexer/multiplexer.

Memory Folding and Operator Definition The third and final phase of the modelling process consists of the folding of all multiple references to memory locations. This leads to a model in which *global* paths and the alternative paths of a demultiplexer/multiplexer represent functionalities that can be mapped to hardware operators. The hierarchically organized cells are expanded and simplified in a bottom-up order. Since a HMC can be referenced more than once in the final model, simplification precedes inclusion.

The HMC of `alu.action` serves as an example. Paths that do not contain references to the memory are left unaltered. Firstly, the sequencing edge between the write to `L1` and the read from `R` can be removed because it was only necessary to order the writes to `L1` and `L2`. Since these are disjoint, there is no need for sequencing. Secondly, the common occurences of `L1`, `L2` and `L3` are merged, leading to a more compact flow.[9] There are two kinds of

[8] The `switch` statements are essentially treated the same way.

[9] A simple optimization can be performed along the way: Whenever a sequence of a demultiplexer and a multiplexer contains only identical alternatives the sequence can be replaced safely by one of these alternatives. As a special case, this removes single signals or sequencing edges, i.e. empty conditionals.

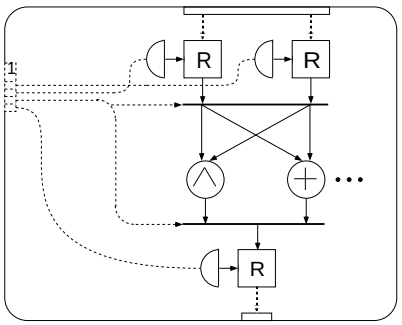

Figure 3. The final result for `alu.action` after flattening and optimizations.

memories:

- latches serving as temporary repositories during the execution of one instruction and
- true memories (or delays) storing values between the execution of two instructions.

A memory is identified as a latch if there is no possibility to route a value to it and retrieve it in another cycle. Formally, this means that no edge in the transitive hull of the sequencing graph leads from a reading access to a writing access. It follows that all values read from a latch were stored there in the same instruction cycle. Thus, they can be accessed more directly by circumventing the latch.

Performing all these transformations bottom-up results in a graph in which all latches are only written to, but never read. They can then trivially be removed. The final result for `alu.action` is shown in figure 3.

The remaining conditional scopes are used to define operators. Everything between the corresponding demultiplexer/multiplexer pair is considered to be executable on one operator which is controlled by the pair's decision making connections.

Conclusion and Future Work

We have shown how an `nML` instruction set description can be transformed into a hardware model. Our method is based on powerful data representations and the concept of unifying the behavioural model and the encoding model. We are currently evaluating the algorithm to determine how efficient the

generated architectures are. The addition of further attributes to `nML` should make it possible to give the designer a finer control of the transformation process. Future activities will include the modeling of machines with partly specified encodings and the formalization of the design library that is used to specify what operators can be merged.

References

[1] M. Freericks. The nML Machine Description Formalism. Technical Report 1991/15, Technische Universität Berlin, Fachbereich 20, Informatik, Berlin, 1991.

[2] A. Fauth, A. Knoll. Automated generation of DSP program development tools using a machine description formalism. Proceedings ICASSP 93, Minneapolis, Minn., April 1993.

[3] D. Lanneer, G. Goossens, F. Catthoor, M. Pauwels, H. De Man. An Object-Oriented Framework supporting the full High-Level Synthesis Trajectory. Proceedings CHDL 91, Marseille, France, April 1991.

[4] P. Marwedel. MSSV: Tree-Based Mapping of Algorithms to Predefined Structures (Extended Version). Technical Report No. 431, University of Dortmund, 1993.

[5] A. Fauth, A. Knoll. Translating signal flowcharts into microcode for custom digital signal processors. to be presented at ICSP 93.

[6] F. Löhr. Erstellung eines Werkzeugs zur automatischen Generierung eines Simulators aus einer abstrakten Maschinenbeschreibung (in German). Studienarbeit. Technische Universität Berlin, August 1992.

[7] G. Meyer-Berg. Hand – The Architectural Compiler Front End for CBC. Esprit 2260 Technical Report CBC.a/Siemens/Y4-M3/1, January 1992.

A. Fauth, M. Freericks
Technische Universität Berlin
Institut für Technische Informatik
Franklinstr. 28/29
10587 Berlin
Germany
`fauth@dasburo.de`
`mfx@cs.tu-berlin.de`

A. Knoll
Universität Bielefeld
Technische Fakultät
Postfach 100131
33501 Bielefeld
Germany
`knoll@techfak.uni-bielefeld.de`