AUTOMATED GENERATION OF DSP PROGRAM DEVELOPMENT TOOLS USING A MACHINE DESCRIPTION FORMALISM+

A. Fauth and A. Knoll

Technische Universität Berlin
Sekr. FR 2-2
D 1000 Berlin 10
Germany
fauth@cs.tu-berlin.de

ABSTRACT

We introduce a retargetable microcode generator for application specific digital signal processors (ASDSPs). The primary goal of our work is to quickly provide system architects with the set of tools necessary for program development (assemblers, instruction set simulators, debuggers and compilers); in particular when the processor architecture is refined simultaneously with the algorithm. After a modification of the architecture, only the machine description written in our language nML must be altered, the tools are then produced automatically. The machine description need not explicitly list every possible instruction in full length. Instead, a derivation tree is described. Through the extensive use of inheritance and sharing of properties, this description can be very compact. Based on the latter, the recognition of critical data paths and the analysis of machine inherent parallelism is solely performed by the tool generator.

1 INTRODUCTION

Digital signal processors (DSPs) are specialized microprocessors designed for real-time manipulation of digital signals. Applications for these processors include telecommunications, control, robotics, encryption and audio/video consumer electronics. Typical calculations performed in digital signal processing are based upon the computation of vector or matrix products, convolution of time-series, pattern recognition and solution of linear equation systems. Moreover, decision making is omnipresent; this gives rise to the need for microprogrammable architectures with flexible branch controllers or possibilities to execute operations depending on a condition. Most microprogrammable DSP architectures also provide for special addressing modes, e.g., based on modulo-n and bit-reversed address arithmetic.

To achieve the required performance, DSPs feature a higher degree of parallelism than standard microprocessors. They are usually designed after the Harvard architecture model incorporating several separate memory banks and buses. This characteristic feature is also visible to the programmer. The building blocks of the processor all work in parallel and must be programmed separately, ensuring a conflict-free program behavior. Furthermore, DSPs often feature a multi-stage instruction and data pipeline, which can be a source of conflicts. The pipelines make most instructions seem to be completed within one machine cycle.

The optimizations typically performed in a product design cycle, have proven to be insufficient in the context of DSP application development because they presuppose a static architecture and only transform the algorithm. Not only is there a need for high throughput in the real-time environments, but the cost of chip real estate must also be brought down to a minimum. Another constraint is the limit for processor power consumption especially in portable systems. It is therefore of paramount importance that the processors and the programs be highly optimized. Iterative refinement of hard- and software very often leads to somewhat arcane processor structures.

Currently, when a new processor is developed, the corresponding program development tools are also designed more or less from scratch. This procedure, however, is very tedious and error-prone. It does not seem to be adequate in highly competitive environments.[1] The alternative is to provide the chip with a more sophisticated functionality than needed for the specific application. While this is obviously more costly than a lean design, our approach allows for easy adaptation of the tools and provides for an instant feedback to the developer.

The demands traditionally imposed on code generation especially hold for ASDSP code generation in that all processor features should be fully exploited by the compiler. This means that the generated code must be correct, of high quality and efficiently use the resources. By contrast, short turnaround times are not the main objective. The code generation task is user-guided to guarantee a high degree of flexibility.

In this paper essentials of the architectural front-end analyzing the architecture description and thus allowing for easy retargeting are presented. A more detailed look at aspects of code generation can be found in [1].

+ Part of this research is supported by the ESPRIT 2260 ("SPRITE") project of the European Community.

[1] The same holds for the usual retargeting process of a high level language compiler. "Easy" retargetability in that context often means that it is possible to reuse parts of the compiler or that not everything has to adapted by hand. It does not mean that a compiler can be automatically tuned to a new, (slightly) different machine within a short time.

0-7803-0946-4/93 $3.00

2 SYSTEM OVERVIEW

The general layout of the system is depicted in figure 1. The major input to the retargeting process is a description of the target processor written in nML [2].

The first step in retargeting the *compiler* is the analysis of the (modified) machine description. Here, all architectural details essential for code generation but *not* explicitly provided by the user are identified. The working model of the architecture *hereby synthesized* does not necessarily correspond in a one-to-one manner to the actual hardware but mimics it closely enough. Hardware operators and a network of interconnections between them are extracted from the machine description. Target *machine dependent* code generation modules are generated, which model available operation chains and resource conflicts by providing a set of patterns and combination possibilities. These modules are linked with *machine independent* modules which include reusable optimizations and parameterized expansion rules. This completes the retargeting process.

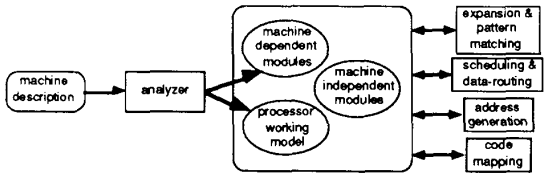

Figure 1 System Overview

For the *simulator/debugger* and the *assembler*, retargeting is even simpler because all the information necessary is directly contained in the machine description and most of the analysis is omitted.

In our environment, all aspects of the *behavior* of the system that are important for proper translation must be specified. This presupposes the precise modeling of word lengths and numerical effects (overflow and underflow conditions as well as saturation modes) in the processor specification and in the description of the application algorithm.

The algorithm is produced by a schematic editor and presented in a signal flowchart. This flowchart can be translated directly to a simple single-assignment language thereby preserving the whole parallelism of the chart. The actual generation of microcode is based on the Cathedral 2nd framework [3] and starts with a front-end translating a flowchart into an intermediate data/control flow graph representation. In a second step, some machine independent tasks are performed (including memory management, constant folding and condition optimizations). Operations which are not directly executable on the machine are then expanded. For the expansion task the mechanism provided by [4] is used. Then, a pattern-matcher is invoked for code selection and local operation chaining. After matching the graph of the application program, the microoperations are scheduled. For the scheduling task, a list-scheduler, as reported in [5], is incorporated. In the final stages, symbolic names are mapped to data memory and basic blocks are mapped to program memory. The desired output (assembly source or binary image) is generated.

3 MACHINE DESCRIPTIONS

The machine description language nML [2] is intended for describing arbitrary single instruction stream architectures. Such architectures feature a single program counter, but can otherwise consist of an unlimited number of building blocks, such as AGUs and ALUs. The abstraction level that nML aims at is the *programmer's model* of the processor, i.e., the instruction set and the memory model. Therefore, most implementation details of the actual machine are hidden. A single nML description can be shared among a number of applications which have a need for a formal description, making nML a suitable input language for a wide range of system development tools. The nML machine model is very simple: A running machine executes a *program* consisting of a thread of *instructions*. While executing one instruction, a *program counter (PC)* points to the instruction executed next. The machine has a *state* stored in *memory locations*. The sole purpose of a program is to change the contents of these memory locations. Each instruction is therefore modeled as a transition function from state to state. Instruction semantics are given as a sequence of *assignments* which are similar to traditionally known *register transfers*. The program flow may be changed by writing to the PC location. To specify a machine using nML, precise knowledge is required about

- all existing storage classes including the register files,
- all data formats directly available on the machine,
- all occurrences of alignment restrictions,
- the exact semantics of instructions including side-effects,
- the complete set of addressing modes,
- the usage of condition codes,
- the possibilities of program flow control and
- internal data-processing structures like pipelines.

Complex architectures may allow hundreds of legal combinations of operations and addressing modes to compose the instruction set. If instructions have side-effects, exhaustive descriptions might grow very large. Description size is therefore reduced by *sharing* similarities among a variety of instructions. An nML description is a file consisting of an attributed grammar and assorted definitions. For nML grammars, all *nonterminals* must have derivations. There must be no cycles. This implies that all strings having no productions consist only of *terminals*.

```
op instruction = jump | aluOp | ...

op aluOp (a:aluAction,s1:SRC,s2:SRC,d:DST)
  action = { tmp1 = s1; tmp2 = s2;
             a.aluAction; d = tmp2; }
  syntax = format("%s %s,…",a.syntax,…)
  image  = format("…

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

mode SRC = REG | ...

mode REG (n:card (3)) = R[n]
  syntax = format ("D%d",n)
  image  = format ("%3b",n)
```

Figure 2 Machine description example

Two derivation possibilities exist:

- *or-rules* branching into alternatives and
- *and-rules* merging shared properties.

Basically, nonterminals exist for four purposes (the corresponding keywords are given in parenthesis):

- data representations (`type`)
- storage classes (`mem`)
- operations (`op`)
- addressing modes (`mode`)

Each terminal string produced by the grammar corresponds to one member of the instruction set. By deriving all terminal strings, the complete instruction set can be listed. By itself, however, such a string contains no useful information. All semantic aspects are held in *attributes*. A fixed set of attributes is given for each nonterminal. Attributes have *expressions* as their definitions. Expressions are either arbitrary "`C`"-like expressions or sequences of statements. Both kinds may contain references to attributes of parameters to the and-rule.

Three attributes are predefined:

- `action` describes the run-time semantics of an operation,
- `syntax` holds its assembly language syntax and
- `image` contains the values to set instruction word fields.

`nML` in its present form is considered as a core language that can be extended if need be by adding application specific attributes. Figure 3 shows a fragment of an `nML` description.

4 ARCHITECTURAL ANALYSIS

The goal of the analysis steps is to identify all architectural details needed for a specific tool but which are not directly provided by the machine description. For code generation, the underlying machine (this includes the processor kernel *and* dedicated hardware) is described by its *execution properties*, i.e., the operations it can execute, and its *topology*, i.e., the possible data routes. As in [6] a *microoperation* (*MO*) is the most primitive entity to model activity on the machine. Its semantics is represented by a (machine independent) *named quintuple* $N=(I, O, U, T, F)$ with

- N – the name of the MO
- I – the set of input resources to the MO (source operands)
- O – the set of output resources to the MO (destination)
- U – the set of functional units participating in the execution
- T – the number of clock cycles required for execution
- F – the instruction word field-code pairs that encode the MO

A *microinstruction* (*MI*) is a set of conflict-free microoperations which are started at the beginning of the same machine cycle. Conflicts can occur with respect to I, O, U and F. The generation of conflict-free MIs is one of the main tasks to code generation. The result of the analysis steps is a model for

- the executable microoperations as named quintuples,
- the hardware operators depicting the processor,
- the netlist interconnecting the operators and
- patterns for conflict-free MOs.

The patterns represent conflict-free MO clusters which can be executed in one cycle on interconnected components. Horizontal coding of machine instructions introduces further restrictions. These are also captured in the patterns by providing only legal (i.e. encodeable) chains. Combination possibilities of patterns are modeled via operator assignments which impose additional resource conflicts on the scheduling task.

4.1 Flattening the Derivation Tree

An enumeration of all described instructions is achieved by *flattening* the derivation tree of the machine description. This process traverses the tree and visits all possible productions deriving all possible terminal strings. Each produced terminal string represents a *fully expanded instruction*[2] (FEI). An FEI defines all actions taking place on the processor during one instruction cycle; no additional (for code generation relevant) semantic action can be performed except the listed ones.

To enumerate the complete instruction set, all rules are *calculated*. This means that referenced attributes are substituted by their value. If the definition of an attribute includes a condition or a switch, *condition hoisting* is performed producing all possible instantiations of the current rule. The same is done with all or-rules.

After flattening, the `action`-attribute of every FEI is processed: First, data dependent expressions are concatenated into trees. After this, every FEI's `action` is represented by a set of expression trees (a forest). Then, two different analysis procedures are started. The purpose of the first is to extract a set of hardware operators; the purpose of the second is the generation of code patterns.

4.2 Mimicking the Datapath

After the expression trees for all FEIs have been constructed, a model for the topology of the architecture must be found. A correspondence between operations and operators is built. Each `nML` primitive function (like + or <<) that is a node in an expression tree is associated with a *functional block* or *operator* of the processor. Since every operator is assumed to possibly execute more than one operation, the goal is to minimize the forest of expression trees by *collapsing* trees by introducing merge and split nodes (see figure 3).

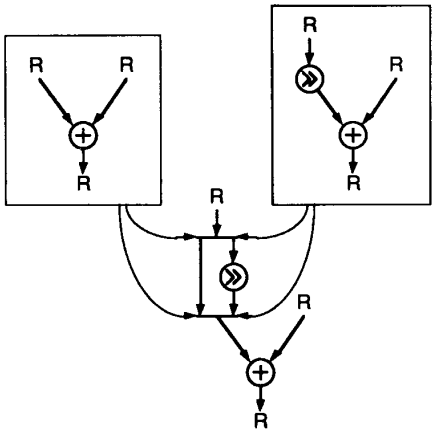

Figure 3 Expression tree collapsing

[2] Such a fully expanded instruction may still be parametric in many bits. It is obviously not desirable to produce all instances of an instruction that includes a multi-bit literal yielding operations like `add #1,Ri`, `add #2,Ri`, etc.

The number of functional blocks is minimized in this step yielding more powerful blocks. Two trees can be collapsed if they both include an *identical* operation with different operands or include operations which are *similar* (like addition and subtraction).

The merge nodes represent multiplexers and the split nodes demultiplexers. Links to the split, merge and operator nodes are held for every operation because these represent the set of used functional units. Figure 3 shows an example

The trees can be simplified even further. A set of a split node, a merge node and operator nodes (this may also be a no-op) between them can be merged into a macro-operator node if all operations that have a link to either the split or the merge node actually have a link to both.

For every operation the sets I, O, and U can now be given.

4.3 Automated Pattern Generation

An important phase of code generation performs *operation selection* and *chaining* with pattern matching [7][8]. This process finds partial instructions on the machine that implement the algorithm, but leaves the high degree of freedom to the scheduler. The necessary patterns are generated from the FEIs. Problems would occur if the expressions were directly taken as patterns when the machine has

- a large amount of internal parallelism and/or a large number of orthogonal addressing modes:

  The algorithm must be carefully ordered to make efficient use of independent functional units. Since each pattern matched describes a specific combination of all sub-instructions, further reordering (i.e. scheduling) is precluded.
- operators that can be eluded or bypassed or operations with side effects:

  The patterns cannot be matched directly to the algorithm because they are *overdetermined*. Consider a combined `shift-add` instruction. By shifting zero bits, a pure `add` operation can be performed, while by adding a zero, a pure `shift` is done. Since it cannot be assured that shifts and adds are coupled in the algorithm, the pattern-matcher will fail to select code.

Therefore, the expression trees will only serve as a base for the construction of the pattern base. The basic idea is to split the trees up and form patterns out of the parts. These patterns can then be matched separately onto the algorithm and put together in the subsequent scheduling phase. It must be assured that either all combinations are legal or that restrictions are modeled by resource conflicts.

A splittable tree is seen as a cluster of (at least two) microoperations. These clusters can be divided:

- *Orthogonality criterion*

  For each pair of clusters $I = A_I \oplus B_I$ and $J = A_J \oplus B_J$ it is true that the two clusters $K_1 = A_I \oplus B_J$ and $K_2 = A_J \oplus B_I$ exist. This is typically the case for totally independent parts of the machine that do not even share fields in the instruction word (like a bus allowing data transfers in parallel to an ALU executing various numerical computations).
- *Operator pass modes criterion*

  No-ops $A_\varnothing$ or $B_\varnothing$ exist such that for a cluster $I = A_I \oplus B_I$ there are clusters $K_1 = A_\varnothing \oplus B_I$ and $K_2 = A_I \oplus B_\varnothing$. Of course, the operations A_I and B_I must be logically independent, i.e. data dependencies are not allowed. Such no-ops exist for various functional units and are commonly called pass-modes. They can be explicit operational modes of the unit or identity functions like addition with zero, multiplication by one or a shift by a value of zero.

These conditions guarantee that all instructions may be split into (at least two) operations and be recombined, possibly at the expense of efficiency if no-ops have to be inserted.

The generated patterns are dumped in a human-readable format such that experienced users can manually optimize them or add patterns which the analysis failed to detect.

5 CONCLUSION

We have shown how a DSP architecture can be described in a convenient way by using an easy-to-read machine description language. Tools have been developed which use this description as input and automatically produce the program development tools assembler, simulator/debugger and compiler. The purpose is to drastically simplify the adaptation of development tools when the architecture of a processor is changed during its design phase. The analysis steps performed during the tool generation ascertain that the code produced by the generated tools is highly efficient. The toolset we have developed is in its prototype stage. Our future efforts will concentrate on the refinement of the presented techniques especially for handling side-effects. Other important tasks are the development of different scheduling techniques, solutions for graph pattern matching and the design of tools for displaying design statistics, hardware operator loads and other useful information.

REFERENCES

[1] A. Fauth, A. Knoll "Automatic Generation of DSP Program Development Tools Utilizing a Machine Description Formalism", Technischer Bericht 1992-31, Technische Universität Berlin, FB 20, Berlin, 1992

[2] M. Freericks, "The `nML` Machine Description Formalism", Technischer Bericht 1991-15, Technische Universität Berlin, FB 20, Berlin, 1991

[3] D. Lanneer et al., "Open-ended System for High-Level Synthesis of Flexible Signal Processors", Proceedings EDAC 90, 1990

[4] D. Lanneer et al, "An Object-Oriented Framework supporting the full High-Level Synthesis Trajectory", Proceedings CHDL 91, 1991

[5] R. Hartmann, "Combined scheduling and data routing for programmable ASIC Systems", Proceedings EDAC 92, 1992

[6] D. Landskov et al., "Local Microcode Compaction Techniques" Computing Surveys 12 (2), 1980

[7] M. Ganapathi et al., "Retargetable Compiler Code Generation", Computing Surveys 14 (4), 1982

[8] C.W. Fraser et al., "BURG – Fast Optimal Instruction Selection and Tree Parsing", ACM SIGPLAN Not. 27 (4), 1992