Abstract

We describe an approach to hardware/software co-design that starts with a high-level specification of a target machine and a synchronous data flow representation of an algorithm. The instruction set description is translated into a netlist-level machine description. A set of independent tools successively transform the algorithm into a program for the target processor. We employ the machine description formalism nML, in which a processor architecture is defined solely by its instruction set semantics. Modularization and sharing of semantic properties between instructions are modelled by structuring the complete description as an attributed grammar. Analysis tools guide the user in optimizing both the hardware and the software. The design trajectory is explained by using the ADPCM algorithm as an example application and a core DSP as initial target machine.

Implementation of Complex DSP Systems Using High-Level Design Tools

Markus FREERICKS, Andreas FAUTH and Alois KNOLL[†] *

September 6, 1995

1 Introduction

Today's complex DSP algorithms are often implemented as custom- and semi-custom VLSI circuits. Hardware synthesis [1] can generate a combination of hard- and software directly from an algorithmic system specification. Current synthesis tools [2, 3, 4] can handle small to medium-sized algorithms (some hundred operation nodes, simple control flow) or they are specialized for a highly regular structure (e.g. video processing, specialized memory architecture[5]).

For more complex and decision-oriented applications, customizable DSP cores are often used. Such a core combines a basically fixed general-purpose "CPU" kernel with extensions in the form of application-specific accelerator data-paths. Thus, the cost of hand-optimizing the core design can be shared amongst different applications; new hardware is only needed to cope with the "hot spots" of an algorithm.

When designing such an architecture and its application-specific extensions, a design methodology has to be created that encompasses the specification, test, and implementation of both software and hardware. "Classic" synthesis environments are not well suited for this task, because they are based upon the assumption that the hardware can be modified. Furthermore, the hardware is described as a netlist built up from register-level hardware entities, which can be allocated and connected at will.

Our framework[6] is based upon the nML[7] machine description formalism, in which a processor is described solely by its instruction set. For each instruction, its exact semantics are given at the register-transfer level. There is no explicit controller description; instead, the designer specifies an instruction encoding from which a controller can be derived.

By creating a design abstraction at the instruction set level, rapid prototyping of DSP core architectures becomes feasible: given a retargetable code generator, the system designer can compile application benchmarks for an architecture, analyze the utilization of its components and modify the high-level machine description easily. All algorithm development and machine-independent simulation is done at the signal-flow level, in a dedicated DSP language like SILAGE[8] or in a high-level specification language like ALDiSP[9]. Machine-level simulation can be performed with instruction set simulators generated from the nML description. If special optimizations are needed (e.g. when the code generator does not fully utilize a dedicated hardware unit), the designer can manually intervene at any level of the compilation process.

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

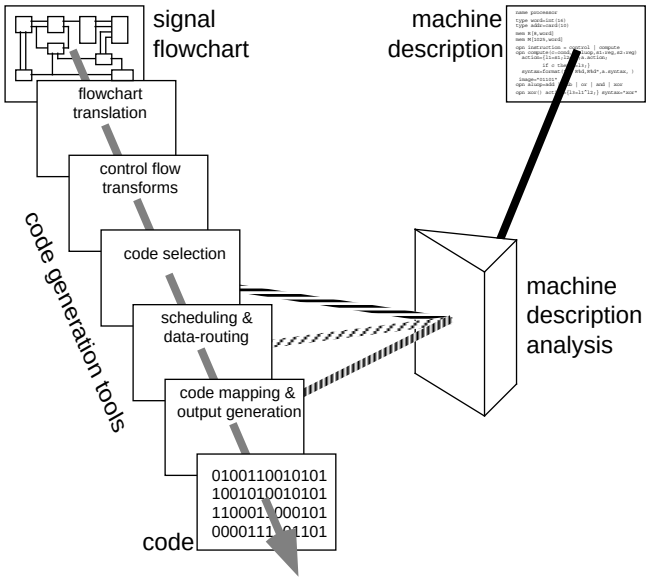

Figure 1: Design Trajectory

2 System Overview

An overview of our system's design trajectory is shown is Fig.1. The algorithm is supplied in the form of a flowchart, which is transformed into the internal control/data-flow graph (CDFG) representation. This representation is human-readable and common to all tools. The target architecture is specified as an nML description. From this, a machine analyzer creates tool-specific machine description files (since each tool needs different "aspects" of the machine description).

In the following, each tool will be described as it is used in the design process. We start with a machine description.

3 Initial nML Description

As an (arbitrary) initial target architecture, we use a 16-bit core that consists of RAM, four registers, and a 4-operation ALU (cf. Fig.2). Each input to the ALU is routed through a modifier that can negate, set to zero, or shift by a constant amount. The output of the ALU is compared with zero and the CZ flag is set accordingly. Additions also generate a carry bit. In parallel to each non-jump instruction, a move to or from memory can be executed. There is only one jump instruction, which is conditionally controlled by an arbitrary PSW flag.

The nML description for this machine is depicted in Fig.3. This 63-line text suffices to completely describe the behaviour of our hardware. The attributes that describe binary encoding or textual representation of the assembly code are left out for clarity; they are only needed in the last stage of the compiler.

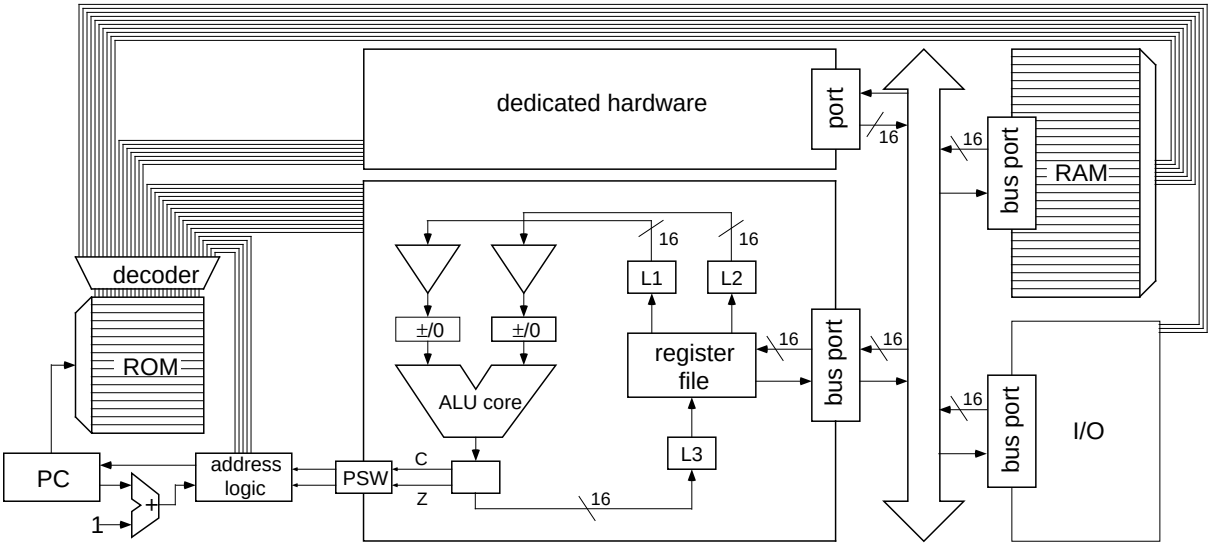

Figure 2: The Processor Core

The nML description is structured as a grammar of twenty nonterminals, the root of which is `instruction`. Nonterminals are either defined via alternatives ("|"-rules) or by parameterized definitions that may refer to both terminals (literal values) and instantiations of other nonterminals. Each nonterminal can define attributes that refer to its parameters and their attributes. The instruction set is defined as the set of all possible derivations of the instruction tree; each instruction's semantics is given by the value of its `action` attribute. Note that nML imposes no restriction on the set of possible attributes; descriptions can be extended with arbitrary attributes (e.g., timing or cost-related) if there are tools that recognize them. Our current framework supports attributes for semantics, instruction encoding, and assembly language representation.

4 The Retargeting Process

Each time the user modifies the nML description, the framework must be "retargeted", i.e. all machine-dependent files have to be updated. Retargeting is performed by an analysis tool that extracts the specific information that each machine-dependent tool needs. Some tools, especially those that realize the classic "software compilation" optimizations (common-subexpression elimination, constant-folding, conditional scope optimization, etc.) are almost entirely machine-independent. There are three main stages in our compiler that depend heavily upon specifics of the target architecture: expansion, chaining, and scheduling.

```
01 type addr = [0..511]    \\ program size
02 type word = int(16)     \\ word size
03 type midx = [0..1023]   \\ main mem size
04 type ridx = [0..3]      \\ register set size
05 type flidx= [0..15]     \\ bits in PSW
06 type shift= [-8..7]     \\ shift range
07
08 mem PC[1,addr]          \\ program counter
09 mem M[midx,word]        \\ main mem size
10 mem R[ridx,word]        \\ register set size
11 mem PSW[flidx,bit]      \\ status word
12
13 mem CC[1,bit] alias=PSW[0] \\ carry bit
14 mem CZ[1,bit] alias=PSW[1] \\ zero bit
15
16 mem L[3,word]           \\ ALU latches
17 mem BUS[1,word]         \\ bus latch
18
19 mem NORM[1,word] alias=M[0] \\ memory-
20 mem EXP [1,word] alias=M[1] \\ mapped
21 mem MANT[1,word] alias=M[2] \\ accel.
22
23 op instruction = jump | alu_and_move
24
25 op jump(c:bit, ci:flidx, target:addr)
26 action={if PSW[ci] == c then PC = target;}
27
28 op alu_and_move(a:aluop, m:moveop)
29  action={ m.pre; a.action; m.post;}
30
31 op moveop = load | store
32
33 op load(from:midx, to:ridx)
34  pre= {BUS = M[from];}
35  post={R[to] = BUS;}
36
37 op store(from:ridx; to:midx)
38  pre= {BUS = R[from];}
39  post={M[to] = BUS;}
40
41 op aluop(a:alu; o1,o2,d:ridx; l:modl; r:modr)
42  action={L[0] = R[o1]; l.action();
43          L[1] = R[o2]; r.action();
44          a.action;
45          CZ = (L[2]==0);
46          R[d] = L[2];}
47
48 op alu = add | and | or | xor
49
50 op add() action = {CC::L[2]=L[0] + L[1];}
51 op and() action = {    L[2]=L[0] & L[1];}
52 op or()  action = {    L[2]=L[0] | L[1];}
53 op xor() action = {    L[2]=L[0] ^ L[1];}
54
55 op modl = shift_l | neg_L | zero_l
56 op shift_l(s:shift) action={L[0]= L[0]<<s;}
57 op neg_l()          action={L[0]= -L[0];}
58 op zero_l()         action={L[0]= 0;}
59
60 op modr = shift_r | neg_R | zero_r
61 op shift_r(s:shift) action={L[1]= L[1]<<s;}
62 op neg_r()          action={L[1]= -L[1];}
63 op zero_r()         action={L[1]= 0;}
```

Figure 3: Processor Core Instruction Set

4.1 Expansion

The *expansion* phase maps the "abstract" operations of the initial application algorithm to the "concrete" set of operations that is implemented by the target architecture. To give an example: our processor core has no hardware multiplier; for each multiplication that occurs in the algorithm, the expansion tool must find a suitable implementation in terms of the availabe hardware operators (+,<<, and conditional jumps).

This phase is based on a library that provides the replacement rules for all operations that can not be directly implemented. A large set of these rules is machine-independent and predefined; the machine dependency lies in marking those operations available in hardware, and specifying expansion policies for those operations with multiple implementations of differing cost. The library can also be extended by hand to provide for "special-purpose" operations implemented by accelerator paths.

4.2 Chaining

The *chaining* phase [10] contracts groups of connected operations into data-path operations that can be executed on one data-path within one instruction cycle. On our example architecture, combinations of shift/negate/zero followed by add/and/or/xor operations provide chaining opportunities. Chaining implements part of the instruction selection task of ordinary compilers; it is based on a library of pattern matching rules. This library has to be generated in toto from the nML description.

4.3 Scheduling and Routing

The last phase consists of *scheduling* and *data routing*: the partial instructions that are generated in the chaining phase must be ordered in time; signals must be routed through registers and memory locations. Scheduling and routing are strongly related tasks: the scheduler tries to minimize the register life-times of signals; the data routing algorithm must determine which values are kept in registers according to their probable life-times. As scheduling is NP-complete, heuristics play an important role. We employ a list scheduler guided by an adaptable multi-level priority function.

This phase needs a detailed resource model of the target machine, a library of the valid transfer operations, and resource usage information encoded in reservation tables for the chained instructions.

5 The Application Algorithm

As our example algorithm, we use a subset of the ADPCM (Adaptive Digital Pulse Code Modulation) algorithm, which is employed in telecommunication applications. In our framework, algorithms are represented as control-data-flow graphs (CDFGs), i.e. synchronous data-flow graphs with control edges. Operation nodes include the standard arithmetic functions, a "select" operator and type conversion operators. Later stages of the compiler generate additional transfer- and jump-operations, as well as signal attributes (such as "lifetime" and "location"). Conditions are modelled by "scopes", which are more flexible than standard "basic block" control flow models. The sole memory operation is the "delay", which stores a value for one iteration of the algorithm. Each scope is controlled by a condition, and all signals defined in the

scope are valid only if that signal is true. The "select" operation is used to merge the results of different scopes. Represented as a data-flow graph, the whole ADPCM algorithm consist of ca. 2600 operation nodes. Our subset is the "predictor", which contains almost all multiplications. Figure 4 describes from what operations it is made up.

Category	Type	Count	%
ALU op	abs	10	1.91
	add	60	11.47
	mult	8	1.53
	neg	28	5.35
	shift	33	6.31
	sign	25	4.78
	and	1	0.19
	xor	18	3.44
control	eq	7	1.34
	gt	12	2.29
	lt	5	0.96
	not	13	2.48
	select	24	4.59
memory	delay	18	3.44
book-keeping	bundle	49	9.37
	unbundle	46	8.80
	rename	156	29.83
accelerator	norm	10	1.91
sum ALU		193	36.90
sum control		61	11.66
sum total		523	100.00

Figure 4: Initial Application Node Count

Note the high percentage of control operations: the ratio of control-related operations to ALU-operations is nearly 1:3. A naïve compiler would have to generate at least one compare and one jump per compare/select, from which a minimal execution time of 48 cycles can be deduced.

6 Utilization statistics

Our framework includes tools that let the user inspect the CDFG during the different phases of compilation, both by visually presenting the graph and by giving statistics keyed by node categories. These statistics guide the designer when modifications to algorithm and hardware are made. Major points of interest are

- the number of chainings that were found: If only a subset of the chaining patterns is used, the instruction encoding can be tightened; modifiers might be moved to different positions in the data-path to ensure better chaining possibilities.

Category	Type	No HW	Booth	Full Mult
ALU op	add	143	95	95
	neg	46	46	46
	shift	226	78	78
	and	1	1	1
	hwmult			8
	bmult		32	
	or	40	6	6
	xor	18	18	18
control	setEQ	99	43	43
	setLT	53	62	62
	not	13	13	13
memory	delay	18	18	18
book-keeping	rename	269	137	137
	cast	307	101	101
accelerator	norm	20	20	20
sum		1253	670	646
sum w/o book-keeping		677	549	525
nodes after chaining		1146	686	666
scheduled cycles		807	538	471

Figure 5: Three Multiplier Alternatives

- frequent operation combinations: if certain sequences of operations occur frequently, they can be taken as candidates for accelerator data-paths.
- utilization of operators: depending upon how often they are used, operators may be removed or replaced by cheaper alternatives.

An example of the latter phenomenon is multiplication: our original algorithm contains 8 multiplications of 7-bit values with 14-bit results. Since our core contains no multiplier, these are expanded into shift/add combinations of ca. 24 operations each. The inclusion of a hardware multiplier would thus eliminate ca. 192 nodes from the algorithm. As an experiment, we have specified two accelerator datapaths, one containing a full multiplier, the other implementing a 2-bit Booth step (allowing 8-bit multiplication in 4 steps).

To include the full multiplier, the nML description is extended by two lines:

```
23  op instruction(i:instr)
23a action={i.action; M[0] = M[1] * M[2];}
23b op instr = jump | alu_and_move
```

This models a memory-mapped multiplier that runs once per cycle.[1] Figure 5 shows the node counts of the three alternatives after expansion and chaining, and the total length of the algorithm after scheduling.

[1] The specification for the Booth multiplier is not shown because it is basically the same, only somewhat larger.

7 Conclusion

We have presented a retargetable code-generation framework that can be used to optimize both hardware- and software-components of a DSP application. A concise machine description formalism serves as the sole input language and facilitates an easy retargeting process. For a large application algorithm (full ADPCM), our system takes about 3 hours of runtime, most of it is spent in the scheduling phase. To estimate the quality of the generated code, we compared it against a hand-coded version of the ADPCM algorithm for an architecture similar to our core (with Booth-step accelerator) that had a total of ca. 1500 instructions. Based on this, we estimate that the code produced by our compiler is ca. 30% slower than hand-written code.

Current work is concerned with modelling pipelined data-paths and complex memory models. [11] shows how an nML model can be translated into a hardware-level net list; we are now working on an extension to nML in which we can provide a "net skeleton" that guides this process. Finally, we are considering the generation of code for multi-processor systems.

References

[1] **M. C. McFarland, A. C. Parker, R. Camposano (1990):** The High-Level Synthesis of Digital Systems, *in: Proc. of the IEEE, Vol. 78, No. 2., pp.301-318, 1990*

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

[3] **F. Catthoor (1992):** Design Methodologies for application-specific signal processing architectures, *Tutorial presented at EUSIPCO 92*

[4] **M. Potkonjak, J. Rabaey (1993):** Exploring the Algorithmic Design Space using High Level Synthesis, *in: Eggermont et.al (eds): VLSI Signal Processing VI, IEEE Special Publications, pp. 123-131*

[5] **P.E.R. Lippens, J.L. van Meerbergen, W.F.J. Verhaegh, A.E.van der Werft (1993):** Modular design and hierarchical abstraction in Phideo, *VLSI Signal Processing, VI, Eggermont et.al. (eds), IEEE Signal Processing Society, 1993*

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

[7] **M. Freericks (1991):** The nML Machine Description Formalism, *Technical Report 1991/15, Technische Universität Berlin, Fachbereich 20, Informatik, Berlin, 1991*

[8] **Mentor Graphics/EDC (1991):** Silage User's and Reference Manual,

[9] **M. Freericks, A. Knoll, L. Dooley (1992):** The Real-Time Programming Language ALDiSP-0: Informal Introduction and Formal Semantics, *Forschungsberichte des Fachbereichs Informatik Nr.92-26, TU Berlin*

[10] **A. Fauth, G. Hommel, C. Müller, A. Knoll (1994):** Global Code Selection for Directed Acyclic Graphs, *5th International Conference on Compiler Construction (CC'94), LNCS 786, pp. 128-142*

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