Instruction-Set Modelling for ASIP Code Generation

Rainer Leupers, Peter Marwedel
University of Dortmund
Department of Computer Science XII
44221 Dortmund, Germany
email: *leupers|marwedel@ls12.informatik.uni-dortmund.de*

Abstract

A main objective in code generation for ASIPs is to develop retargetable compilers in order to permit exploration of different architectural alternatives within short turnaround time. Retargetability requires that the compiler is supplied with a formal description of the target processor. This description is usually transformed into an internal instruction set model, on which the actual code generation operates. In this contribution we analyze the demands on instruction set models for retargetable code generation, and we present a formal instruction set model which meets these demands. Compared to previous work, it covers a broad range of instruction formats and includes a detailed view of inter-instruction restrictions. [1]

1 Introduction

ASIPs (application-specific instruction set processors) can be regarded as hardware components bridging the gap between ASICs and general-purpose processors. One major bottleneck in ASIP-based design is *code generation*, which is either done manually or by using compilers. Manual code generation at the assembly level is both time-consuming and error-prone, and reuse of software is hardly possible. On the other hand, compiling code from a high-level language description to a number of different ASIPs requires sophisticated compilers capable of efficiently mapping algorithms to varying target processors. Therefore, *retargetable compilation* has received attention in the CAD research community. An overview of recent techniques is given in [2].

Usually, the compiler transforms a processor description into an internal instruction set model, on which the actual code generation operates. In the first place, the style of such an instruction set model depends on the *controller architecture* of an ASIP. Kifli [3] describes different options for ASIP controller architectures which are currently used. The most commonly used controller architecture is the *programmable microcoded controller* (PMC, fig. 1), due to its high speed efficiency and low design effort. The basic operation of a PMC is the execution of one or more *register transfers* (RTs) in each machine cycle. Therefore, the granularity of an instruction set model should be the RT-level.

The purpose of this paper is to introduce an instruction set model for ASIPs with PMC architecture, which supports retargetable compilation. The main advantage of this model is that a broad range of possible instruction formats is covered, thereby permitting a high degree of retargetability. The model is an essential feature of the retargetable compiler RECORD, aiming at ASIPs in the DSP domain [4, 5].

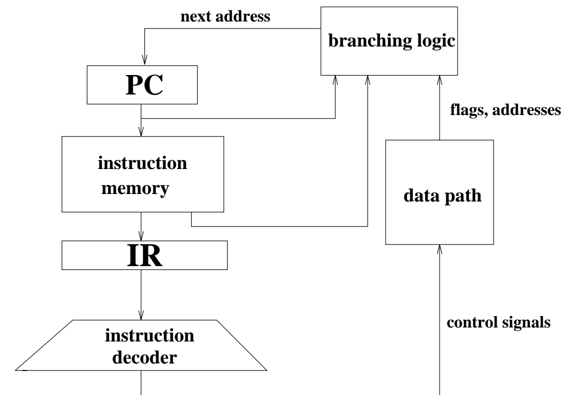

Figure 1: *Programmable microcoded controller (PMC) architecture for ASIPs. The instruction register (IR) and the instruction decoder are optional.*

The organization of this paper is as follows: In section 2 we analyze the demands on instruction set models which arise from contemporary ASIP architectures. Section 3 discusses to what extent these demands are met by previous work in the area of microprogramming and retargetable compilation. Our new instruction set model is formally described in section 4, and its application in practice is exemplified in section 5.

2 Demands on ASIP instruction set models

Essentially, an instruction set model suited for retargetable compilation should be capable of covering a broad range of possible instruction formats. We describe demands on instruction set models by means of RTs. In particular, a versatile model should reflect the

[1]Publication: 9th International Conference on VLSI Design, Bangalore/India, Jan. 1996, ©IEEE

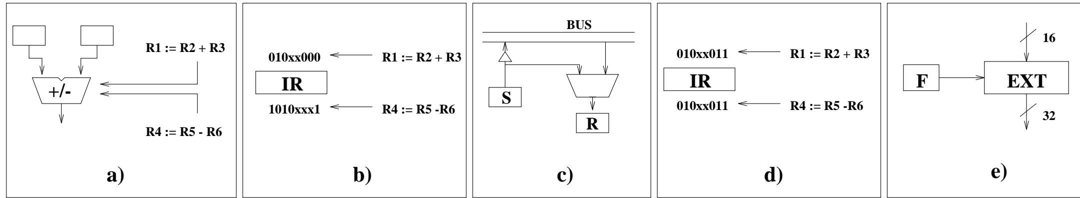

Figure 2: *a) Resource conflict: Both operators in the RTs are mapped to the same ALU in different modes. b) Instruction set conflict: Two RTs require incompatible opcodes in the instruction register IR. c) Alternative versions: Two possible data routes exist for moving data from register S to R. d) Side effect: Both RTs have to same partial control word and thereby imply each other. e) Residual control: The state of flag F decides whether component EXT performs sign or zero extension on its input data.*

followings items, which arise from instruction formats of realistic ASIP architectures:

Resource conflicts: (Fig. 2 a) RTs, which use the same resource in different modes may not be executed within the same machine cycle. For instance, an ALU will perform either addition or subtraction at a time, and a register can only load one certain value in each instruction cycle.

Instruction set conflicts: (Fig. 2 b) The *binary encoding* of the control word may prohibit simultaneous execution of two resource-compatible RTs. In many processors, such encodings are used to decrease the control word length.

Alternative versions: (Fig. 2 c) For each RT, there may exist alternative *code versions*, i.e. different partial control word settings which trigger execution of that RT, e.g. due to different data routes through a processor's datapath with same sources and destination, or application of algebraic rules like commutativity of ALU operators.

Side effects: (Fig. 2 d) Different RTs may share partial control word settings, i.e. execution of one RT necessarily implies another different RT. Whenever a register contains a live value, its destruction by a side effect must be prevented during code generation. In other cases it may be favorable to exploit the side effect in order to increase parallelism or to just tolerate it.

Residual control: (Fig. 2 e) Control signals are usually assumed to be originating at the instruction memory or register. In case that certain control signals are likely to change only rarely, they may be relocated to *residual control registers* in order to minimize the control word length. Therefore, the instruction set model not only has to account for partial control words associated with RTs, but also for the required *machine state* regarding the residual control registers.

3 Related work

Instruction set models for microcoded controllers have been proposed earlier in the area of microprogramming. Davidson et al. [6] present a detailed model for microoperations intended to be used for experimental evaluation of different microcode compaction techniques. The model includes multi-cycle operations but is restricted to VLIW machines, i.e. instruction set conflicts cannot be represented. The same holds for residual control.

Different approaches to ASIP modelling have also been introduced in recent work on code generation for embedded processors. The MSSQ compiler [7] has no explicit instruction set model, but internally uses a graph model of the target processor. A special processor description style is required, and residual control registers are excluded. Possible side effects are prevented by generation of partial control word settings which disable unused storages for each instruction cycle. Alternative versions and inter-instruction conflicts are handled during a code compaction phase. In CHESS [8] a similar graph model is employed, which is constructed from a mixed structural/behavioral processor description in the nML language. The CHESS model also accounts for inter-instruction conflicts and alternative versions, but residual control and side effects are not considered. The CodeSyn system [9] reads C-like behavioral specifications in terms of *instruction patterns*. However, the instruction patterns and register classifications cannot be generated from a more general model, and handling of side effects and residual control is not reported.

Completely different approaches to instruction-set modelling are taken by researchers who focus on very high code quality requirements such as [10, 11], which currently only handle restricted instruction formats.

4 The instruction set model

The proposed instruction set model is based on the notion of register transfers (RTs) and "no-operations" (NOPs). RTs describe transfer of values from and to registers, memories, and external processor ports. NOPs are used for handling of side effects as mentioned in section 2.

4.1 Execution conditions

RTs are executed only under certain *execution conditions*, which are represented by Boolean functions in our model. As explained in section 2, control signals in general may have two origins: the instruction memory/register and residual control registers. In order to treat both kinds of control signals in a uniform way, we

define a set of Boolean variables representing all control signals:

Definition: Let w be the control word length and $RES = \{R_1, \ldots, R_k\}$ be the (possibly empty) set of residual control registers of a given ASIP. Let $B(R_i)$ denote the bitwidth of register R_i. The *execution condition variables* are defined as the set of Boolean variables

$$ECV = \{v_i | i = 0 \ldots w-1\} \quad \cup$$
$$\{v_{ij} | R_i \in RES, j = 0 \ldots B(R_i) - 1\}$$

Thus, each variable $v \in ECV$ represents one instruction bit or one residual control register bit. An execution condition for an RT or NOP requires a certain setting of those variables:

Definition: Let $r = |ECV|$. An *execution condition* is a Boolean function $F : \{0,1\}^r \rightarrow \{0,1\}$

Example: Suppose, an RT requires a partial control word setting (`x` denotes don't care):

$$v_7 v_6 v_5 v_4 v_3 v_2 v_1 v_0 = \texttt{10xx01x1}$$

and bit no. 3 of residual control register R_1 needs to be zero. Then, its execution condition is

$$F = v_7 \cdot \overline{v_6} \cdot \overline{v_3} \cdot v_2 \cdot v_0 \cdot \overline{v_{13}}$$

In case of alternative code versions, the execution conditions in general have the form

$$F = P_1 + \ldots + P_k$$

where the P_i's are product terms on ECV. During the code compaction phase of code generation, alternative versions need to be explicitly computed from the execution conditions. In order to exclude consideration of don't care control signals, we define:

Definition: Let $PI(F) = \{p_1, \ldots, p_k\}$ denote the set of *prime implicants* for an execution condition F. The *version set* for F is the set of Boolean functions $\{F_1, \ldots, F_k\} : \{0,1\}^r \rightarrow \{0,1\}$, which correspond to $PI(F)$, i.e. $F_i(b) = p_i(b)$ for all bindings b of execution condition variables.

By computing the version sets all unnecessary restrictions are removed, which increases the freedom for code compaction. Although the number of prime implicants of a Boolean function may be exponential in the number of variables, the effort for version computation remains reasonable in practice: In case of VLIW instruction formats with long instruction words, only a few control signals are not don't care for each RT. In case of heavily encoded instruction formats, don't care signals are few, but the instruction word length is small.

RTs assign values to destinations under certain execution conditions, while NOPs disable sequential components during one instruction cycle. Let SEQ denote the set of sequential processor components (registers and memories), and P_{OUT} the set of processor output ports.

Definition: A *register transfer* is a triple $RT = (d, v, VS)$, where $d \in SEQ \cup P_{OUT}$ is the *destination*, v is a *value*, and VS is a version set.
A *no-operation* is a pair $NOP = (d, VS)$, where $d \in SEQ$ is the *destination* and VS is a version set.

As in most other approaches, we use *expression trees* for representation of values. For code selection using expression trees, there exist efficient algorithms based on dynamic programming [12]. Therefore, representation of values in RTs is not further discussed in this paper.

Like RTs, NOPs are only active under a certain execution condition. NOPs have to be activated during code generation, whenever live values need to be preserved for consumption in a later machine cycle. Most approaches to instruction-set modelling for ASIPs do not consider NOPs explicitly, but assume that NOPs are implicit in the encodings of available RTs. However, in case of VLIW instruction formats, each processor component has separate control lines, and the compiler must generate code that explicitly sets these control lines in order to prevent undesired side effects. Obviously, there is a relation between RTs and NOPs:

Remark: For a destination $d \in SEQ$, let $R_d = \{RT_1, \ldots, RT_n\}$ be the set of RTs which write to d, and $S = \{F_1, \ldots, F_k\}$ the union of the version sets of R_d. Then, the NOP for d has the execution condition

$$F = \overline{\bigvee_{i \in \{1,\ldots,k\}} F_i}$$

This relation permits explicit computation of NOPs, once the set of all RTs is known. The instruction set model for a given ASIP comprises the union of all RTs and NOPs. During code generation, binary code for instructions can be emitted by deriving those control bit settings, for which an execution condition is true. Since all versions of execution conditions are kept in the model, it permits postponing parts of the code selection phase to the code compaction phase, in which an appropriate version can be selected.

5 Example

This section demonstrates the capabilities of the above instruction set model using a real-life processor from the DSP domain: The instruction format of Texas Instruments' TMS320C25 DSP [13] incorporates all of the peculiarities mentioned in section 2, and therefore may serve as a "worst-case" example. The TMS320C25 has a heavily encoded 16-bit instruction word and comprises several residual control registers such as `PM` for certain *product modes*, or `OVM` for activation of *saturating arithmetic*. We use some of its total of 133 instructions for exemplifying how the different aspects of the instruction set are captured in the model (table 1). Instead of noting versions as Boolean functions F, we use the bitstrings for the corresponding instruction word and residual control register settings, i.e. those settings for which F becomes true. [2] Due to the concept of versions, **resource and instruction set**

[2] `x` denotes don't care, `c` denotes an arbitrary but fixed setting, e.g. for immediate constants. `PR`, `TR`, `BO`, `PM`, `ACCU`, `AR`, `ARP` denote TMS320C25 registers and register files.

No.	RT/NOP	version(s)
(1)	`PR := TR * sign_ext(imm.const.)`	101ccccccccccccc
(2)	`PR := B0[AR[ARP]] * B0[AR[ARP]]`	001110011ccccccc
(3)	`PM := imm.const.`	11001110000010cc
(4)	`PR := TR * B0[AR[ARP]]`	110011111ccccccc, 001110001ccccccc, 001110111ccccccc
(5)	`ACCU := ACCU - PR`	00111011xxxxxxxx
(6)	`NOP ACCU`	xx11x000xxxxxxxx
(7)	`ACCU := ACCU + (PR SHR 6)`	00111001xxxxxxxx $\wedge$ `PM = 11`

Table 1: *Some operations and versions in the TMS320C25 DSP instruction set model*

conflicts cannot be distinguished in our model. Both are reflected by incompatibility of versions. RTs no. (1) and (2) of table 1 write to the same destination `PR` and therefore have a resource conflict. RTs (2) and (3) are resource-compatible, but have an instruction set conflict due to contradicting partial instruction word settings.

RT no. (4) has three **alternative versions** which arise from macro-instructions of the TMS320C25. Version 3 of RT (4) has a **side effect** on register `ACCU`: RT (5) will also be executed. When RT (4) is generated during code generation, selection of its appropriate version depends on the current context: When RT (5) happens to be also generated, version 3 will execute both RTs in parallel. In contrast, when `ACCU` contains a live value that must not be destroyed, version 2 should be selected, which also executes NOP (6) for destination `ACCU`. In this way, version selection during code generation may enhance the code quality by increasing the parallelism. The RECORD compiler uses an Integer Programming model for exploitation of alternative versions and prevention of undesired side effects during code compaction [5].

RT no. (7) exemplifies **residual control**. Execution of that RT requires the 2-bit product mode register `PM` to have the binary value `11`. In order to select RT (7), code must be generated which loads `PM` in an ealier machine cycle. For instance, RT (3) may serve this purpose.

Although the total number of TMS320C25 operations becomes relatively large in our model (approx. 2500 compared to 133), it is constructed within only 30 CPU seconds on a SPARC-20, using the technique described in [4].

6 Conclusion

A new instruction set model for ASIPs with a programmable microcoded controller architecture was presented. Compared to previous approaches, the model is capable of capturing a broad range of instruction formats and peculiarities in the instruction set, such as side effects and residual control. This makes it suitable for usage in a retargetable code generator, which must handle varying processor architectures and instruction formats. By making alternative code versions for register transfers and no-operations explicit, a means of phase coupling between code selection and code compaction is provided, permitting to achieve higher code quality. Furthermore, the model can be automatically generated from a HDL processor description.

References

[1] M. Strik, J. van Meerbergen: *Efficient Code Generation for In-House DSP Cores*, European Design & Test Conference (ED & TC), 1995, pp. 244 – 249

[2] P. Marwedel, G. Goossens (eds.): *Code generation for embedded processors*, Kluwer Academic Publishers, June 1995

[3] A. Kifli, G. Goossens, H. De Man: *A Unified Scheduling Model for High-Level Synthesis and Code Generation*, European Design & Test Conference (ED & TC), 1995, pp. 234 – 238

[4] R. Leupers, P. Marwedel: *A BDD-based frontend for retargetable compilers*, European Design & Test Conference (ED & TC), 1995, pp. 239 – 243

[5] R. Leupers, P. Marwedel: *Time-constrained Code Compaction for DSPs*, 8th International Symposium on System Synthesis (ISSS), 1995

[6] S. Davidson, D. Landskov, B. D. Shriver, P. W. Mallet: *Some experiments in local microcode compaction for horizontal machines*, IEEE Trans. on Computers, vol. 30, No. 7, 1981, pp. 460 – 477

[7] L. Nowak: *Graph Based Retargetable Microcode Compilation in the MIMOLA Design System*, 20th Annual Microprogramming Workshop (MICRO-20), 1987, pp. 126 – 132

[8] J. van Praet, G. Goossens, D. Lanneer, H. De Man: *Instruction Set Definition and Instruction Selection for ASIPs*, 7th Int. Symp. on High-Level Synthesis, 1994, pp. 11 – 16

[9] C. Liem, T. May, P.G. Paulin: *Instruction Set Matching and Selection for DSP and ASIP Code Generation*, European Design & Test Conference (ED & TC), 1994, pp. 31 – 37

[10] T. Wilson, G. Grewal, D. K. Banerji: *An integrated approach to retargetable code generation*, 7th Int. Symp. on High-Level Synthesis, 1994, pp. 70 – 75

[11] A. H. Timmer, M. T. J. Strik, J. L. van Meerbergen, J. A. G. Jess: *Conflict Modelling and Instruction Scheduling in Code Generation for In-House DSP Cores*, 32nd Design Automation Conference (DAC), 1995, pp. 593 – 598

[12] A. Aho, M. Ganapathi: *Code Generation using Tree Matching and Dynamic Programming*, ACM Trans. on Programming Languages and Systems, vo. 11, no. 4, 1989, pp. 491 – 516

[13] TMS320C2x User's Guide, Rev. B, Texas Instruments, 1990