Practical Instruction Set Design and Compiler Retargetability Using Static Resource Models

Qin Zhao[1], Bart Mesman[1,2] and Twan Basten[1]
[1] Eindhoven University of Technology
P.O. Box 513, NL-5600 MB Eindhoven, The Netherlands
[2] Philips Research Laboratories
Prof. Holstlaan 4, NL-5656 AA Eindhoven, The Netherlands

Abstract

The design of application (-domain) specific instruction-set processors (ASIPs), optimized for code size, has traditionally been accompanied by the necessity to program assembly, at least for the performance critical parts of the application. The highly encoded instruction sets simply lack the orthogonal structure present in e.g. VLIW processors, that allows efficient compilation. This lack of efficient compilation tools has also severely hampered the design space exploration of code-size efficient instruction sets, and correspondingly, their tuning to the application domain. In [13] a practical method is demonstrated to model a broad class of highly encoded instruction sets in terms of virtual resources easily interpreted by classic resource constrained schedulers (such as the popular list-scheduling algorithm), thereby allowing efficient compilation with well understood compilation tools. In this paper we will demonstrate the suitability of this model to also enable instruction set design (-space exploration) with a simple, well-understood and proven method long used in the High-Level Synthesis (HLS) of ASICs. A small case study proves the practical applicability of the method.

1 Introduction

Application (-domain) specific instruction-set processors (ASIPs) are becoming increasingly popular in systems on a chip (SoCs) because they offer the possibility to exploit the characteristics of the application (-domain) to gain considerable savings in silicon area, power consumption, and code size. There are roughly three ways to tune an ASIP core to an application, that can be used in combination:

- By synthesizing a communication (busses) and storage (registers) infra-structure, just sufficient for the application.
- By hardware acceleration [9]. Functional units are added to the data path that perform relatively coarse grain functions characteristic of the application, for example a butterfly unit in an FFT processor.
- By minimizing the width of the instructions required to control a given data path [12]. One way is to encode frequently occurring (sequences of) operations with short instruction words. Another possibility is to limit the number of instructions by restricting certain combinations of operations that the data path can execute in parallel.

Hardware acceleration potentially offers the largest benefits on all accounts, especially for applications that contain much regularity. It is also the most complex method for the designer because it requires changes in the communication and storage hardware and the design of the dedicated functional units. The mentioned ways to tune the ASIP core all have the same severe drawback: They add the necessity to 'recognize' in the application, instructions supported in the tuned instruction set. Often this task of recognition (code selection) is performed by the programmer himself, either by writing assembly or with the use of APIs in the source code to call the dedicated instructions or hardware. Both require a lot of source code rewriting and low-level programming, and both are detrimental for the maintainability and portability of the code. Alternatively, the compiler contains a code selection phase that recognizes valid instructions in the Data Flow Graph (DFG), often using pattern matching and graph covering techniques [4], [5]. Unfortunately the results of applying these techniques have been quite disappointing. They are for a large part responsible for the considerable overhead in both schedule length and code size (reported in the order of 800% [7]) of compiler generated code compared to manually written assembly for ASIPs.

A way out of this status quo is offered by the Static Resource Model (SRM) [13] approach. The SRM approach

targets instruction sets that are minimized by restricting certain combinations of operations that the data path can execute in parallel. Instruction sets in this class, which contains e.g. the so-called issue slot machines, can be modeled in terms of *virtual* resources, easily interpreted by classic resource constrained schedulers such as the popular list-scheduling algorithm. This alternative machine model with virtual resources therefore allows efficient resource constrained compilation with well understood and widely available compilation tools, rather than the poorly performing compilers based on instruction selection.

The SRM approach also enables instruction set design (-space exploration) with an equally well-understood and proven method long used in the High-Level Synthesis (HLS) of ASICs [1]. This method, illustrated in Figure 1, analyzes the time critical loops for bottlenecks in the availability of processor resources required to obtain the target schedule throughput. These bottlenecks can be identified by scheduling the loop and examining the *load diagrams* of the functional resources. The load on critical resources is then relieved by allocating additional resources. The potential use of this method for instruction set design is based on the observation that both the availability of real functional resources **and** virtual resources can be adapted when considering the SRM of an instruction set. Increasing the availability of virtual resources results in an extension of the instruction set. We presume therefore that the SRM view on an instruction set allows instruction set design in terms of allocating resources just sufficient to efficiently execute the critical loops.

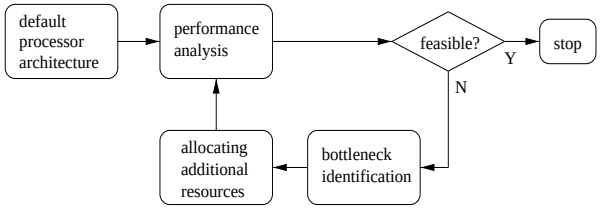


Figure 1. *Design flow of ASICs*

The resulting instruction set can be implemented as an ASIC or an ASIP. ASIPs can be reprogrammed after fabrication. In order to tune the instruction set of a fabricated ASIP to an application, the instruction decoder should be reconfigurable to a certain degree. Note that the real (non-virtual) resources are considered fixed in this case.

This paper is organized as follows. Section 2 presents the basic definitions. Section 3 addresses the problem of constructing an SRM of an instruction set. In section 4 we propose an instruction design flow based on the SRM approach. In Section 5, a small case study is presented. Conclusions and future work are discussed in Section 6.

2 Definitions

A DSP algorithm can be expressed as a data flow graph, which describes the primitive operations performed in an algorithm and the dependencies between those operations.

Definition 1. A *data flow graph (DFG)* is a tuple (V,E), where V is the set of vertices (operations) and $E \subseteq V \times V$ is the set of precedence edges.

In this paper, we consider software pipelined loops. The *initiation interval* (*II*) is the period in between two successive executions of the loop. In a processor architecture, a functional resource can be used in different ways. E.g., a functional resource *ALU* can execute an operation *add* or a *subtract*, etc. For reasons of complexity, we do not wish to enumerate all possible uses of a functional resource in an instruction set. Therefore, we consider the collection of these uses of a functional resource and associate with it an *operation type*. The set of operation types is denoted as T. An *instruction* is now defined as a combination of operation types that can be executed in a single clock cycle. For example, instruction $[add, mul]$ performs the operation type *add* and *mul* in parallel.

An operation type can appear multiple times in an instruction. For an operation type op in an instruction I, we denote this number by $I(op)$. If for two instructions I_0 and I_1, $I_0(op)$ is always at most equal to $I_1(op)$ for each operation type op, we say that instruction I_0 is *contained* in instruction I_1. In this paper we consider instruction sets IS where for each instruction I_0 all contained instructions are also in IS. We call these instruction sets *prefix closed*.

The code generation problem is to find a schedule of a DFG, i.e., to determine a start time for each operation, that satisfies precedence constraints and architectural constraints. These architectural constraints can be modeled either as an instruction set or by introducing functional virtual resources and associating a certain virtual resource usage with each operation. The corresponding resource constraints are *static* in the sense that they only provide a fixed upper limit and any usage of the resources within the limit is valid. We say that the problem has a static resource model.

Definition 2. A *Static Resource Model (SRM)* is a model for generating static resource constraints, where

$-R$ is a set of resources,

$-\tau$ is a function defining the resources in R that an operation needs,

$-\#r$ denotes the number of instances of each resource.

For example, in Figure 4 (b), the set of resources R is listed. Together with function τ as in Figure 4 (c), it models the instruction set in 4 (d).

In general, operation types can be associated with axes and instructions can be geometrically represented as points in the multi-dimensional space as depicted in Figure 4. In this space, instruction I_1 $[add, add, mul, shift]$ is drawn as

point p7, with coordinates $I_1(add)$, $I_1(mul)$ and $I_1(shift)$. The set of points representing the IS can be bounded geometrically (see Figure 4) by a set of planes, called the *convex hull*. Each such plane corresponds to an inequality (Figure 4 (a)) that constrains the parallelism of operations allowed in **any** instruction in IS.

The *convex hull problem* is defined as follows: Given a set of points (valid instructions), construct its convex hull as a set of *m* inequalities:

$$conv(IS) = \{\vec{x} \mid A\vec{x} \leq b\} \quad (1)$$

where $n = |T|$, $A \in R^{m \times n}$ and $b \in R^m$.

In this convex hull, each inequality corresponds to a maximum resource usage in the SRM.

The convex hull problem is a well-known mathematical problem in the field of computational geometry, for which algorithms are available that serve our purposes. These algorithms are not relevant to our research, so we will not discuss them.

3 The Static Resource Model

In this section, we illustrate our method for constructing an SRM for an instruction set. It is a generalization of the method in [13].

3.1 Advantage of the static resource model

As we mentioned before, code generation for ASIP cores makes it necessary to recognize valid instructions in the DFG. This task is referred to as *instruction selection*. It is performed by covering the DFG with valid instructions, such as depicted on the left hand side of Figure 2.

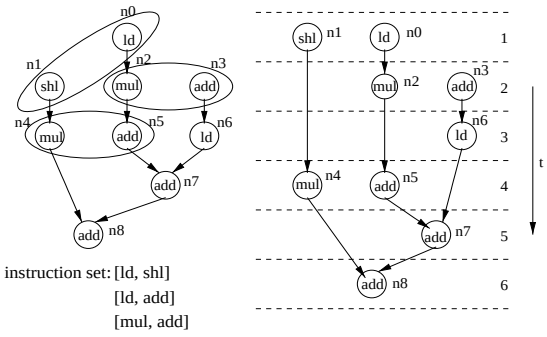


Figure 2. *Instruction selection prior to scheduling may yield inferior results*

The main issue introduced by highly encoded instruction sets is the issue of *phase coupling*: On the one hand, if instruction selection is performed prior to scheduling, the optimal schedule can easily be eliminated as the result of the choices made during instruction selection. On the other hand, if scheduling is performed first, the available instructions may not be able to implement the schedule. Traditional methods perform the tasks in different phases, thereby yielding inferior schedule. This is depicted in Figure 2. The DFG on the left hand side has been covered with machine instructions. The associated schedule (6 clock cycles) for this selection of instructions is given on the right hand side of Figure 2.

The merit of the SRM approach is that by transferring the instruction set constraints to static resource constraints, explicit instruction selection is avoided and the scheduler has the opportunity to generate better schedules in terms of timing and register requirements. This is depicted in Figure 3. We see that the constraints from the instruction set $\{[ld, shl], [ld, add], [mul, add]\}$ can be expressed as three inequalities in Figure 3 (a), which are translated into virtual resources *LM*, *MS*, *SA* directly. The number of instances of each virtual resource is 1. In addition, each operation type uses the virtual resources that it is associated with. For example, *mul* appears in inequality (1) and (2), thus it uses the virtual resources *LM* and *MS*. By applying the resulting SRM of the instruction set to a resource constrained scheduler, we obtain an optimal schedule of 5 clock cycles.

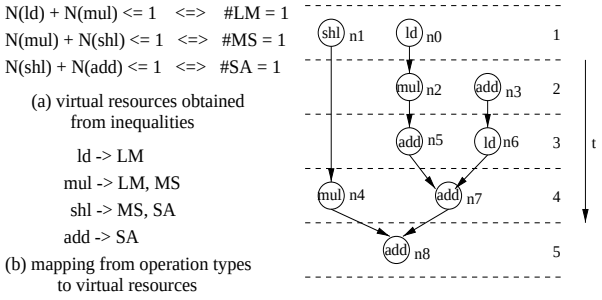


Figure 3. *With an SRM optimal results are obtained*

An example of constructing such an SRM is given in the next subsection.

3.2 An example of constructing an SRM

The approach is illustrated in Figure 4. In this example, for the instruction set given in Figure 4 (d), all the possible instructions are shown as points in the space. The convex hull of the set of points, computed with the cdd package [2], is listed in 4 (a) as a set of inequalities. Notice that inequalities (1) and (2) define the functional resources and inequalities (3) to (5) are translated into virtual resources. For example, inequality (3) is translated into a virtual resource *AM*. For reasons of convenience, we represent a virtual resource by combining the first letters of all those operation types which compose the virtual resource. This will be used

throughout the paper. Figure 4 (b) gives the number of instances of those resources. The mapping from the operation types to those virtual resources is shown in Figure 4 (c). For example, in inequality (4), the coefficient of the second term is 2, which implies operation type *shift* requires the use of two virtual resources *MS* simultaneously.

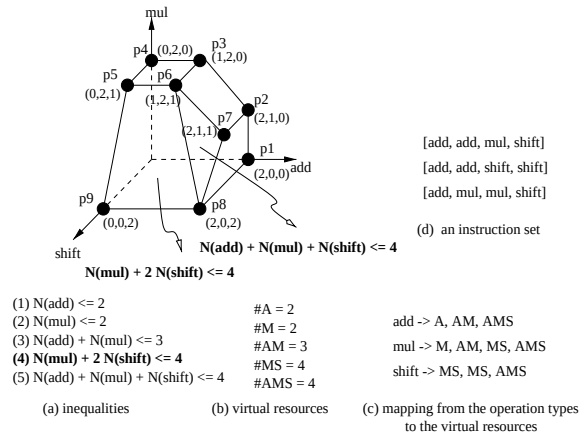


Figure 4. *Construction of an SRM*

4 Optimization of the ISA from the SRM

4.1 Problem statement and approach

Now that we have explained the SRM approach, we will use this approach to perform instruction set architecture (ISA) design.

Usually the instruction set design process is performed independently from the compiler. Thus it could happen that although a good processor architecture is generated, it can not produce the desired performance for applications even if a lot of effort is put on generating an efficient compiler. It is a challenge to design an instruction set for an ASIP that can be encoded using a restricted number of instruction bits, while still offering a sufficient degree of parallelism for critical functions in the target application. We consider the following instruction set design problem.

Problem: Given a set of time critical loop kernels representing an application domain with the corresponding throughput constraints and a target instruction width for the ASIP, design an instruction set and the corresponding SRM such that the throughput constraints can be satisfied.

Noticing the similarity between functional resources and virtual resources, we propose an optimization flow similar to the flow in Figure 1 for allocating functional resources in high-level synthesis.

In this optimization flow, we start with a *default instruction set and processor architecture*. Subsequently, the performance on the critical loops is analyzed, which is explained in more detail in the next subsection. If the performance is insufficient, we look for the responsible (virtual) resources in a step called *bottleneck identification*. The SRM is subsequently modified by e.g. increasing the availability of the (virtual) resources. The essential difference to the high-level synthesis flow in Figure 1 is that we also consider the instruction width as a criterion in the design process to evaluate the modifications made for the SRM.

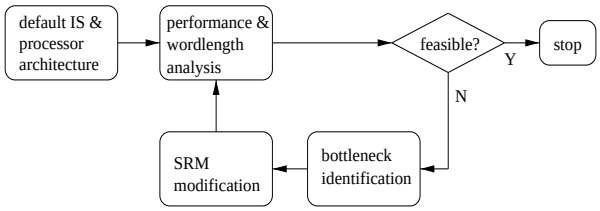


Figure 5. *Overview of the instruction set design flow*

4.2 Performance and bottleneck analysis

Similar to the high-level synthesis approach, performance analysis can be done either fast or accurate. An accurate analysis is obtained by actually scheduling the critical loops and examining the load diagrams. These load diagrams enable the designer to identify critical resources. Alternatively, the performance on the critical loops can be estimated in a fast way by considering a well-known lower bound based on available (virtual) processor resources, which is explained as follows.

Suppose a loop containing 14 *add* operations is mapped on a data path containing three adders. Then we need at least $\lceil \frac{14}{3} \rceil = 5$ clock cycles to execute the loop. By doing this calculation for every available resource, we obtain a lower bound on the initiation interval of a pipelined schedule of the loop. The general experience is that this bound is very tight. The lower bound indicates the critical functional resource in the data path. This estimation can therefore be used for bottleneck identification.

In addition to the allocation of additional resources, in our instruction set design flow we also have the possibility to decrease the resource usage of a critical resource in order to relieve the bottleneck. Therefore it is more convenient to consider the inequalities as in Figure 4 (a), because they describe both the resource availability and, for each operation type, the usage of that resource. For example, inequality (4) indicates that a *mul* uses one instance and a *shift* uses two instances of resource *MS*, of which four are available. Suppose a loop contains 9 *mul* operations and 6 *shift* operations, then inequality (4) determines a lower bound on the initiation interval in the following way. The total resource usage of virtual resource *MS* equals $1 \times 9 + 2 \times 6 = 21$. This results in $\text{II} \geq \lceil \frac{21}{4} \rceil$. In our approach, we consider the inequalities that generate the largest lower bound as the bottleneck for satisfying the performance requirements. The way that

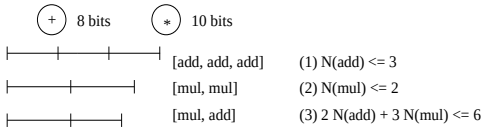
we relieve the bottleneck is explained in the following subsection.

4.3 Modification of the SRM

Suppose that in the previous subsection, inequality (4) in Figure 4 (a) was identified as the bottleneck. We consider two possibilities to modify the SRM.

- Increase the number of instances (the right hand side of the inequality) of virtual resource *MS*. This might result in an increase of the instruction width.
- Decrease the largest usage (the *shift* operation) of the resource *MS*. We consider the largest usage because it has the largest impact on the lower bound.

An example in Figure 7 illustrates the design process. In this example we consider a loop with 6 *add* operations and 9 *mul* operations. The initial instruction set and the corresponding inequalities are shown in Figure 6. *add* is encoded with 8 bits, and *mul* is encoded with 10 bits. The performance requirement of the loop under consideration is given as $\mathrm{II} = 5$.



(a) an instruction set (b) the corresponding inequalities

Figure 6. *An instruction set and the inequalities*

	initial design	modifying right side	modifying left side
inequality	2 N(add) + 3 N(mul) <= 6	2 N(add) + 3 N(mul) <= 7	2 N(add) + 2 N(mul) <= 6
SRM	#A = 3, #M = 2, #AM = 6	#A = 3, #M = 2, #AM = 7	#A = 3, #M = 2, #AM = 6
mapping	add -> A | AM | AM mul -> M | AM | AM | AM	add -> A | AM | AM mul -> M | AM | AM | AM	add -> A | AM | AM mul -> M | AM | AM
IS	[add, add, add] [add, mul] [mul, mul]	[add, add, add] [add, add, mul]	[add, add, add] [add, mul, mul] [add, add, mul]
MII	max(6/3, 9/2, (2*6+3*9)/6) = 7	max(6/3, 9/2, (2*6+3*9)/7) = 6	max(6/3, 9/2, (2*6+2*9)/6) = 5
wordlength	24 bits	28 bits	28 bits

Figure 7. *Modification of the SRM*

In Figure 7, the second column corresponds to the initial design depicted in Figure 6 (b). Virtual resource *AM* with respect to the third inequality is identified as the bottleneck, which lower bounds the II to 7 as calculated in the sixth row. In the third column we evaluate the decision to modify the right hand side of the bottleneck by increasing the number of instances to 7. As a result of this modification, II is now lower bounded by 6. A new instruction $[add, mul, mul]$ is added to the instruction set, thereby increasing the instruction width to 28 bits. In the fourth column we evaluate the decision to modify the left hand side coefficients unevenly by decreasing the larger one. The lower bound on the II is now 5 and the instruction width amounts to 28 bits. From this example, we see the design in fourth column meets the performance requirements, although both design decisions increase the wordlength to 28 bits. We consider a more elaborate example in the next section.

5 Case Study

In this section, we demonstrate the practical applicability of our instruction set design flow using the SRM approach.

		inequlity	SRM	#N	II
[a, m, m]	(1)	N(a) + N(l) <= 3	AL	3	5
[a, s, s, m]	(2)	N(s) + N(l) <= 2	SL	2	5
[a, a, s, l]	(3)	N(m) + 2N(l) <= 2	ML	2	8
[a, a, s, m]	(4)	N(s) + 2N(m) + 3N(l) <= 4	SML	4	8
[a, a, a, m]	(5)	N(a) + 2N(m) + 3N(l) <= 5	AML	5	7
[a, a, a, s, s]	(6)	N(a) + N(s) + 2N(m) + 2N(l) <= 5	ASML	5	7

(a) instruction set (b) SRM and the estimated initiation intervals

Figure 8. *An instruction set and the SRM*

For the instructions given in Figure 8 (a) an SRM can be obtained using the approach in Section 3. We use the fast method for performance evaluation explained in Section 4 rather than performing detailed scheduling. Since the topology of the DFG of the loop is irrelevant for this analysis, we list only the number of operations and their resource usages. We assume the loop contains 9 *add* operations, 3 *sub* operations, 4 *mul* operations and 6 *ld* operations; *add* and *sub* are encoded with 8 bits each, *mul* and *ld* with 16 bits. For reason of convenience, we abbreviate *add*, *sub*, *mul* and *ld* as *a*, *s*, *m* and *l*. The instruction width is constrained to 40 bits. The fourth column gives the number of instances of each virtual resource and the fifth column lists the estimated initiation interval for each virtual resource. Assuming the objective II is 6, this design is far below the performance requirements.

	new inequality	II	extra instructions	WL
3l4l	N(m) + N(l) <= 2 N(s) + 2N(m) + 2N(l) <= 4	5 6	[m, l]	40
3r4r	N(m) + 2N(l) <= 3 N(s) + 2N(m) + 3N(l) <= 5	6 6	[m, l] [s, m, m]	40
3l4r	N(m) + N(l) <= 2 N(s) + 2N(m) + 3N(l) <= 5	5 6	[m, l] [s, m, m]	40
3r4l	N(m) + 2N(l) <= 3 N(s) + 2N(m) + 2N(l) <= 4	6 6	[m, l]	40

Figure 9. *Modification for candidates (3) and (4)*

Figure 9 shows the different results by applying the modification methods in Section 4 to the bottleneck candidates (3) and (4). The first column refers to the design decision

under evaluation. For example, '3l4r' represents the decision to modify the left hand side of inequality (3) and the right hand side of inequality (4). The second column presents the modified inequalities. The third column estimates the II according to the new SRM. Because of the modification, the resource constraints are relieved, and subsequently more instructions are allowed. The fourth column gives the new instructions besides those already provided in Figure 8 (a). The fifth column calculates the wordlength with the new instruction set.

From Figure 9 we can see that all designs sufficiently reduce the bottleneck. We choose the design of rwo three for next iteration because it gives the best combination of performance improvement and extra instructions. The next identified bottlenecks are virtual resources (5) and (6) in Figure 8 and the same procedure is repeated and shown in Figure 10. From this figure, we can see that the second and third design exceed the wordlength limitation and have to be omitted. The first and fourth design meet both the timing and code size constraints and are acceptable.

	new inequality	II	extra instructions	WL
5l6l	N(a) + 2N(m) + 2N(l) <= 5 N(a) + N(s) + 2N(m) + N(l) <= 5	6 6	[s, m, m], [a, m, l]	40
5r6r	N(a) + 2N(m) + 3N(l) <= 6 N(a) + N(s) + 2N(m) + 2N(l) <= 6	6 6	[a, m, l] [a, s, m, m], [a, a, m, m] [a, a, s, s, m], [a, a, a, s, m]	48
5l6r	N(a) + 2N(m) + 2N(l) <= 5 N(a) + N(s) + 2N(m) + 2N(l) <= 6	6 6	[a, m, l] [a, s, m, m] [a, a, s, s, m], [a, a, a, s, m]	48
5r6l	N(a) + 2N(m) + 3N(l) <= 6 N(a) + N(s) + 2N(m) + N(l) <= 5	6 6	[s, m, m], [a, m, l]	40

Figure 10. *Modification for candidates (5) and (6)*

6 Conclusions

In this paper, we propose a method for the design of code size efficient instruction set processors targeted to a certain application (domain). We use the Static Resource Model (SRM) of an instruction set which allows to reason about the instruction set constraints in terms of the limited availability of (virtual) functional resources in the data path of a processor. As a result, efficient compilation can be performed using classic resource constrained scheduling algorithms, instead of the poorly performing code generators that apply explicit instruction selection. The SRM approach also allows instruction set design in terms of allocating (virtual) functional resources, a practical method used in the High-Level Synthesis (HLS) of ASICs. We have described an iterative design flow comprising a bottleneck analysis based on fast performance estimation of the instruction set on a set of performance-critical loops. The availability of critical virtual resources is increased to relieve the bottleneck, resulting in an extension of the instruction set. A small case study demonstrates the practical applicability of the SRM approach to instruction set design. The resulting instruction set is supported by efficient classic resource constrained compilation, thereby preventing the considerable performance overhead introduced by explicit instruction selection, currently used in ASIP compilers.

We intend to extend our work to support more versatile instruction sets, for example an instruction set containing both 16 bits and 32 bits instructions.

References

[1] Frontier Design: A|RT Designer Tutorial.

[2] ftp://ftp.ifor.math.ethz.ch/pub/fukuda/cdd/.

[3] C. Eisenbeis, *Flexible Issue Slot Assignment for VLIW Architectures*, Proc. of the 4th Int. Workshop on Software and Compilers for Embedded Systems, 1999.

[4] C. Liem, T. May and P. Paulin, *Instruction-Set Matching and Selection for DSP and ASIP Code Generation*, Proc. of the Int. Symp. on System Synthesis, 1994.

[5] R. Leupers and P. Marwedel, *Instruction Selection for Embedded DSPs with Complex Instructions*, Proc. of the Europe Design Automation Conference, 1996.

[6] R. Leupers, *Retargetable Code Generation for Digital Signal Processors*, Kluwer Academic Publishers, 1997.

[7] P. Paulin and C. Liem, *Embedded systems: Tools and trends*, tutorial, European Design and Test Conference, Paris, 1996. IEEE Computer Society Press.

[8] B.R. Rau and C.D. Glaeser, *Some scheduling techniques and an easiliy schedulable horizontal architecture for high performance scientific computing*, Proc. of the 14th Workshop on Microprogramming, 1981.

[9] Tensilica Inc., *Application Specific Microprocessor Solution (Date Sheet for Xtensa V1)*, 1998, http://www.tensilica.com/datesheet.pdf.

[10] Texas Instruments: TMS320C62xx CPU and Instruction Set Reference Guide, http://www.ti.com/sc/c6x.

[11] A. Timmer, M. Strik, J. van Meerbergen and J. Jess, *Conflict Modeling and Instruction Scheduling in Code Generation for In-House DSP Cores*, Proc. of the 32nd Design Automation Conference, 1995.

[12] R. Woudsma, *EPICS, A Flexible Approach to Embedded DSP Cores*, Proc. of the Int. Conf. on Signal Processing, Application and Technology, 1994.

[13] Q. Zhao, T. Basten, B. Mesman, K. van Eijk and J. Jess, *Static Resource Models of Instruction Sets*, Proc. of the 14th Int. Symp. on System Systnesis, 2001.