Static resource models for code-size efficient embedded processors

Eindhoven University of Technology Research Report

Eindhoven University of Technology

Department of Electrical Engineering
Eindhoven, The Netherlands

ISSN 0929-8533

STATIC RESOURCE MODELS FOR CODE-SIZE EFFICIENT EMBEDDED PROCESSORS

by

Q. Zhao
B. Mesman
T. Basten

EUT Report 02-E-312
ISBN 90-6144-312-1

Eindhoven
April 2002

CIP-DATA LIBRARY TECHNISCHE UNIVERSITEIT EINDHOVEN

Zhao, Q.

Static resource models for code-size efficient embedded processors / by Q. Zhao, B. Mesman and T. Basten. - Eindhoven : Eindhoven University of Technology, 2002.
(Eindhoven University of Technology research reports ; 02-E-312). - ISBN 90-6144-312-1
NUR 959
Trefw.: ingebedde systemen / compilers.
Subject headings: embedded systems / program compilers / instruction sets.

STATIC RESOURCE MODELS FOR CODE-SIZE EFFICIENT EMBEDDED PROCESSORS[1]

Abstract: Due to an increasing need for flexibility, embedded systems embody more and more programmable processors as their core components. Because of silicon area and power considerations, the corresponding instruction sets are often highly encoded to minimize code size for given performance requirements. This has hampered the development of robust optimizing compilers because the resulting irregular instruction set architectures are far from convenient compiler targets. Among others, they introduce an interdependence between the tasks of instruction selection and scheduling. This so-called *phase coupling* is so strong that, in practice, instruction selection rather than scheduling is responsible for the quality of the schedule, which tends to disappoint. 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 this paper, we present an approach that reduces the need for explicit instruction selection by transferring constraints implied by the instruction set to static resource constraints. All resulting schedules are then guaranteed to correspond to a valid implementation with given instructions. We also demonstrate the suitability of this model to enable instruction set design (-space exploration) with a simple, well-understood and proven method long used in High-Level Synthesis (HLS) of ASICs. Experimental results show the efficacy of our approach.

Keywords: embedded processors, compiler techniques, code generation, instruction selection, instruction set design, constraint analysis, resource modeling, scheduling, phase coupling

Zhao, Q. and B. Mesman, T. Basten
Static resource models for code-size efficient embedded processors.
Eindhoven: Department of Electrical Engineering, Eindhoven University of Technology, 2002.
EUT Report 02-E-312

Address of the author:
Group Information and Communication Systems
Design Technology for Electronic Systems
Department of Electrical Engineering
Eindhoven University of Technology
P.O. Box 513
5600 MB Eindhoven, The Netherlands
Email: q.zhao@tue.nl

[1]This work is partly suported by the IST-2000-30026 project, Ozone.

Contents

Chapter 1

Introduction

Many embedded processors embody programmable processors as their core components. The machine code running on the embedded processors must meet tight timing constraints because of the real-time requirements. Often the program memories reside on chip to obtain small memory latency and low power dissipation. However, the cost of on-chip memories makes it necessary to severely restrict code size. This can be accomplished by exploiting characteristics of the application (domain), allowed by application domain specific instruction set processors (ASIPs) and digital signal processors (DSPs). These processors are frequently chosen as the core processors of systems on a chip due to their ability to balance flexibility and cost. 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) infrastructure just sufficient for the application.
- By hardware acceleration [27]; functional units are added to the data path that perform relatively coarse grain functions characteristic of the application, such as a butterfly unit in an FFT processor.
- By minimizing the width of the instructions required to control a given data path [33]. One way to achieve this 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 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 a Data Flow Graph (DFG), often using pattern matching and graph covering techniques [15], [16]. 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% [24]) of compiler generated code compared to manually written assembly for ASIPs.

The task of code selection, i.e., to *cover* the DFG with processor instructions that implement the individual operations in the DFG, is depicted in Figure 1.1. The main issue introduced by an irregular instruction set is the issue of *phase coupling*: On the one hand, if instruction selection is performed prior to scheduling, the optimal

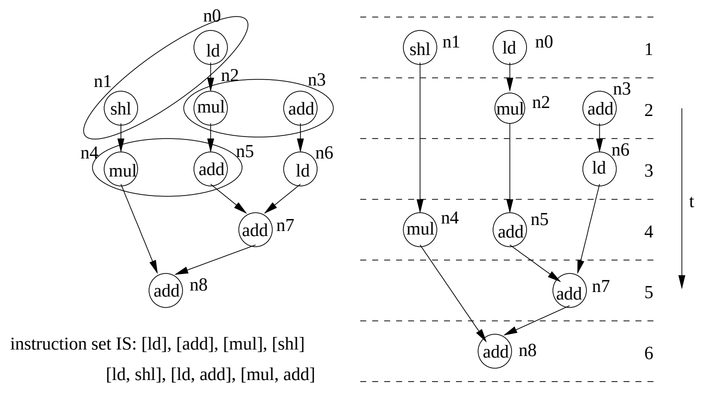

n0
ld
n1
shl
n2
mul
n3
add
n4
mul
n5
add
n6
ld
n7
add
n8
add
instruction set IS: [ld], [add], [mul], [shl]
[ld, shl], [ld, add], [mul, add]
1
2
3
4
5
6
t

Figure 1.1. Instruction selection prior to scheduling may yield inferior results

schedule can easily be eliminated as a 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 which perform these tasks in different phases might yield inferior schedules. This point is illustrated in Figure 1.1. The DFG on the left hand side has been covered with machine instructions which are listed below the DFG. The associated optimal schedule (6 clock cycles) for this selection of instructions is given in the right hand side of Figure 1.1. The imposed instruction selection is rather unfortunate, however, because Figure 1.2 depicts a valid schedule requiring only 5 clock cycles.

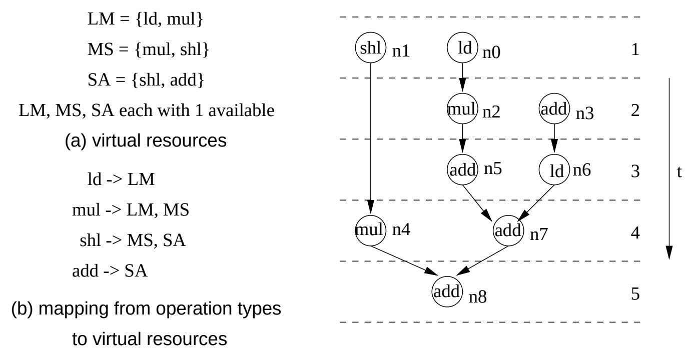


Figure 1.2. With a static resource model optimal results can be obtained

A way out of this status quo is offered by the *Static Resource Model* (SRM) approach [34]. The SRM approach targets the 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 is illustrated in Figure 1.2 for the example in Figure 1.1. The instructions in the instruction set IS have been augmented by the use of the virtual resources $\{ld, mul\}$, $\{mul, shl\}$ and $\{shl, add\}$, which are abbreviated as LM, MS and SA. We have one instance available of each virtual resource. Each operation uses all the virtual resources that it is contained in, e.g., operation mul uses virtual resources LM **and** MS simultaneously. In this way, the LM resource for example models the instruction set constraints that the ld and mul operations may not occur simultaneously. For this so-called static resource model, a list scheduler generates the schedule on the right hand side of Figure 1.2. The reader can verify that the operations at each clock cycle can be implemented with instructions from the original instruction set IS. The resulting schedule has length

5, whereas the schedule in Figure 1.1 has length 6. This example demonstrates that better schedules can be obtained using a static resource model instead of covering the DFG with valid instructions. 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 High-Level Synthesis (HLS) of ASICs [13]. This method, illustrated in Figure 1.3, 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 real functional resources **and** virtual resources can be allocated when considering the SRM of an instruction set. The addition 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.

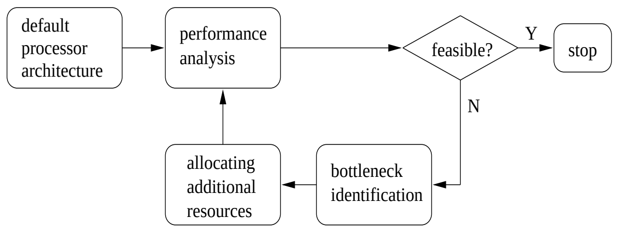

default processor architecture
performance analysis
feasible?
Y
stop
N
bottleneck identification
allocating additional resources

Figure 1.3. 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. Chapter 2 presents related work on instruction set selection and resource conflict modeling of issue slot tables. Chapter 3 details the problem statement and our approach. Chapter 4 addresses the construction of an SRM for an instruction set by using the convex hull approach from the computational geometry field. In Chapter 5, we present the exploitation of instruction set design by evaluating the performance of the application through the SRM approach. In Chapter 6, experimental results are given for benchmarks as well as real applications. Conclusions and future work are discussed in Chapter 7 and 8.

Chapter 2

Related work

The phase of code selection has received a lot of attention in the software compiler community. Traditional compilers use the *template pattern base* to represent the target processor, which essentially enumerates the different partial instructions available in the instruction set. Each partial instruction is represented as a pattern, expressed by means of the intermediate representation of an algorithm. It has been shown that code selection is an NP-complete problem for intermediate representations that take the form of a directed acyclic graph [4]. However, optimal vertical code can be generated in polynomial time when the following conditions are satisfied:

1. the intermediate representation of the algorithm is an *expression tree*;

2. the template pattern base is restricted to contain only *tree patterns*, i.e., it can be represented as a *regular tree grammar*;

3. the processor has a *homogeneous* register structure.

Several code-selector generators are based on a stepwise partitioning of the code selection problem using *dynamic programming*. It is assumed that conditions 1) and 2) of the above mentioned canonical problem are satisfied. In [3] tree pattern matching is done in a bottom-up traversal of the subject tree. For each node, the method computes the minimal cost to cover the subtrees rooted at that node. During the top-down traversal of the subject tree, the tree cover is finally found, by determining the minimal cost at the tree's root node. Dynamic programming is also extended to target heterogeneous register structures, such as Twig [4], Beg [9] and Iburg [12]. Several compilers for embedded processors have adopted dynamic programming for the code selection phase [5, 15, 17]. Such methods often split the DFG into expression trees at the edges representing values being used multiple times and perform tree-covering on each tree separately. Although optimal vertical code can be generated, it cannot guarantee an optimal schedule in the end. In addition, operations have to be inserted among different trees and this will increase the code size. An improved method based on simulated annealing has been described in [19]. Mutation scheduling [22] is another approach considering phase coupling, where algebraic transformations are exploited in order to explore alternative instruction set mappings. A constraint logic programming technique for DSPs with irregular architectures has been presented in [6]. In that approach, all binding decisions are delayed until they are really required, which yields a maximum degree of freedom in code generation. However, this is at the expense of high compilation time.

Another approach for code selection is based on constructing graphs which include complete information of the architecture. In $Chess$ [31], for example, the target processor is described in the nML language [11] and is translated into an Instruction-Set-Graph (ISG), which models connectivity, encoding restrictions and structural hazards. Code selection covers the control-data flow graph with partial instructions (bundles) by searching valid paths in the ISG. All the mentioned methods add a complex step to the compiler chain and eliminate potentially interesting solutions from the schedule and register binding search spaces. In [14, 32], code generation tasks are performed by imposing constraints and the complete solution space is explored while considering all these constraints. However, this approach can only deal with small applications and a restricted set of architectures.

A completely different approach to tackle the code generation problem is to exploit the instruction set constraints statically. Eisenbeis [8] starts from the placement conflicts by using reservation tables for VLIW architectures such as the Trimedia processor. Timmer et al. [29] discusses the modeling of instruction set constraints together with resource constraints for instruction scheduling, but the assumption that all the instruction set constraints can be transferred to resource constraints targets mostly instruction sets with a structure associated with so called issue-slot machines. The virtue of orthogonal instruction sets, like VLIW instruction sets [26], is that they allow to a large extent the modeling of the instruction set constraints by means of a model as presented in [8], [29], or a *static resource model* as proposed in this paper. These models offer the scheduler more opportunity to satisfy the timing and resource constraints than with the use of an explicit instruction selection step. The scheduler rather than the instruction selector is considered the designated place for handling these constraints.

The focus of this paper is on translating the instruction set constraints of highly encoded instruction sets to a static resource model, allowing the use of code size efficient processors in combination with efficient compilation tools based on constraint analysis. Our approach is not restricted to issue slot tables with fixed bit-width for each issue slot; thus it can deal with more flexible encodings than the earlier approaches. In addition, the procedure of constructing a static resource model also provides useful information for instruction set design.

Chapter 3

Problem statement and approach

3.1 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 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*. E.g., on the functional resource ALU, operations *add* and *subtract* can be executed, which are associated with the operation type alu. We denote the set of operation types with T. An *instruction* is now defined as a combination of operation types that can be executed in a single clock cycle.

An operation type can appear multiple times in an instruction (multiple alus can be present in the data path). 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 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 $s(v)$ for each operation $v \in V$, that satisfies precedence constraints and architectural constraints. These architectural constraints can be modeled either as an instruction set or by introducing functional resources and associating a certain 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 within the limit is valid. We say that such a problem has a static resource model.

Definition 2. A *Static Resource Model (SRM)* is a model generating static resource constraints that defines three aspects:

$-$a set of resources R,

$-$a mapping that associates each operation type with the resources in R that it needs, and

$-$the number of instances available of each resource $r \in R$, denoted by $\#r$.

Our approach is motivated by the observation that both resource constraints and instruction set constraints can be expressed by inequalities. For example, if an architecture contains two ALUs and each ALU can be used as an adder or a subtractor, then the resource constraints can be expressed by the following inequality: $N(ALU) \leq 2$, which is equivalent to $N(A) + N(S) \leq 2$, where $N(A)$ and $N(S)$ denote the number of add and subtract operations allowed to execute in parallel. Any schedule satisfying at any time the above inequality indicates a valid resource usage. Similarly, if an instruction set contains instructions $[add, add]$, $[add, sub]$ and $[sub, sub]$, the operation type usage can also be expressed as an inequality: $N(add) + N(sub) \leq 2$, assuming any subinstruction is also a valid instruction.

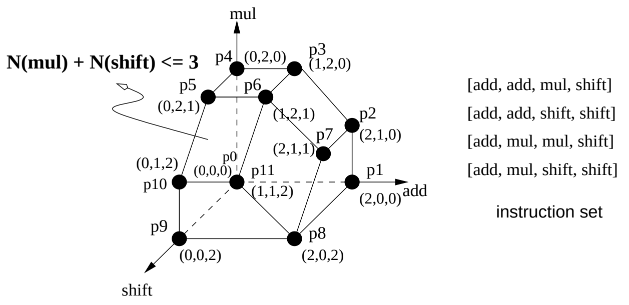


Figure 3.1. Instructions expressed as points in the operation type space

In general, operation types are associated with axes in a multi-dimensional numerical space $\mathcal{R}^d$, where d is the dimension corresponding to the number of operation types, and instructions can be geometrically represented as points in this operation type space. An example is depicted in Figure 3.1. This figure gives an instruction set using three operation types. Only the "maximum" instructions are listed; instructions that are fully contained in other instructions are not explicitly represented. We will do so throughout the paper. Since the instruction set uses three operation types, the operation type space is 3-dimensional. As mentioned, all valid instructions correspond to points in this space. Instruction $I_1 = [add, add, mul, shift]$, for example, is drawn as point $p7$ with coordinates $I_1(add)$, $I_1(mul)$ and $I_1(shift)$. For clarity, only a subset of all the instructions is shown in this figure. Because we assume prefix closedness of instruction sets, the instruction set corresponds to a well-defined, closed subspace of the operation type space. It is our aim to capture this subspace spanned by the instructions via inequalities. Inequality $N(mul) + N(shift) \leq 3$, for example, is one of the inequalities needed to capture this subspace. Inequalities can subsequently be translated to virtual resources. In general, deriving the inequalities and the resulting virtual resources for an instruction set can be perceived as a *convex hull* problem [2] [25]. Here we give some preliminaries of the convex hull problem.

Definition 3. Given a set of points $X = \{x_1, x_2, \ldots, x_k\}$, y is called an *affine combination* of X if y can be expressed as a linear combination of X:

$$y = \sum_{i=1}^{k} \lambda_i x_i \qquad and \qquad \sum_{i=1}^{k} \lambda_i = 1 \tag{3.1}$$

A *convex combination* of X is an affine combination such that each λ_i is non-negative. A *proper* convex combination is one where each λ_i is positive.

Definition 4. A subset S of a d-dimensional space $\mathcal{R}^d$ is called a *convex set* if and only if every convex combination of points in S is also in S. The *convex hull* of a set X, denoted as *conv(X)*, is the set of all convex combinations of X; it is the smallest convex set containing X.

Consider again Figure 3.1. The set of points bounded by the planes $mul = 0$, $add = 0$, $shift = 0$ and the other planes depicted, is the convex hull of the original instruction set IS. As the above example already suggests, a convex set can be described by means of its boundary. In general, such a boundary consists of a set of halfspaces. A halfspace is the set of all points below or above some plane. A halfspace can be seen as a *constraint* on the elements of the set being described and it is defined via an inequality. Consider the example of Figure 3.1 again.

The halfspace containing all points below the plane through points $p5$, $p6$, $p10$ and $p11$ corresponds to inequality $N(mul) + N(shift) \leq 3$. Given a set of halfspaces $\mathcal{H}$ with intersection S, halfspace $H \in \mathcal{H}$ is *non-redundant* if and only if there is some point contained in the intersection of all the halfspaces in $\mathcal{H} \setminus H$ but not contained in S. The notion of non-redundancy is useful in determining the minimal set of inequalities describing convex set S. The halfspace representation of the convex hull of some set X is denoted by $\mathcal{H}(X)$.

It is also possible to describe a convex hull via its *extreme* points. The convex hull in Figure 3.1, for example, can be described via the set $\{p0 = (0,0,0), p1 = (2,0,0), \ldots, p11 = (1,1,2)\}$. Given a convex set S, a point in S is an *extreme* point if and only if it is not a $proper$ convex combination of any two points in S. The notation $\mathcal{V}(X)$ denotes the vertex description of the convex hull of some set X, consisting the set of extreme points.

There are two closely related computational problems concerning the two descriptions of the convex hull of a set X:

- The *vertex enumeration problem* is to compute $\mathcal{V}(X)$ from $\mathcal{H}(X)$.
- The *convex hull problem* is to compute $\mathcal{H}(X)$ from $\mathcal{V}(X)$.

It is known that both problems are polynomially solvable [2]. In code generation for ASIPs and DSPs, deriving virtual resources from a given instruction set can be perceived as a variation of the convex hull problem, since the instructions form the vertex description and the virtual resources form the non-redundant halfspace description. On the other hand, in the instruction set design space exploration, one can optimize the instruction set by modifying the SRM to meet the real-time constraints, and subsequently calculating the corresponding instructions. This can be perceived as a variant of the vertex enumeration problem.

3.2 Problem statement

In this section, we generalize the problem of constructing the SRM for a given instruction set and present the convex hull approach for this problem. In addition, we present an approach for the instruction set design (-space exploration) by tuning the corresponding SRMs.

Problem 1. The general problem can be defined as mapping a given instruction set to an SRM such that any schedule for a DFG satisfying the resource constraints posed by the SRM corresponds to a valid instruction selection. We say that the instruction set *has* an SRM.

We propose a solution strategy based on expressing the instruction set constraints as inequalities, like $N(add) + N(sub) \leq 2$ in the example in Chapter 3.1. We start from an initial instruction set IS. It is followed by a step called *operation-type statistics*, which calculates the number of times operation types as well as their combinations appear in an instruction set. In this way we enumerate all the potential extreme points of the instruction set in the operation type space. We then search for an SRM by computing the convex hull. With the resulting inequalities, we can obtain a new instruction set NIS by enumerating all the instructions allowed under those inequalities. The equivalence of the instruction set constraints and the SRM is verified by comparing the new instruction set NIS to the initial instruction set IS. The SRM itself is derived directly from the inequalities, but is only accurate if the aforementioned test turns out positive. The procedure is illustrated in Figure 3.2. In case the SRM is not accurate, it can be profitable to adapt the instruction set that it fits the SRM, because then all the advantages of the SRM approach can be exploited.

Since the SRM approach can be applied easily to evaluate the performance of an application, especially a loop kernel, it is natural to consider this approach for adapting the designed instruction set to the performance requirements of an application. We generalize the following instruction set design (-space exploration) problem.

Problem 2. Given a set of time critical loop kernels (belonging to a single application) 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.

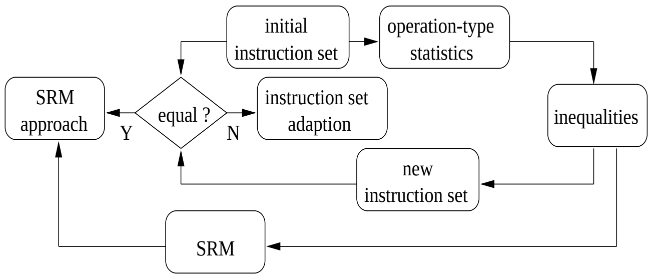


Figure 3.2. Overview of the SRM approach

In Figure 1.3, a practical and commonly used design flow is depicted for allocating functional resources in the context of High-Level Synthesis (HLS). Noticing the similarity between the functional resources and the virtual resources, we propose to apply this flow for allocating *virtual* resources in the context of designing an instruction set.

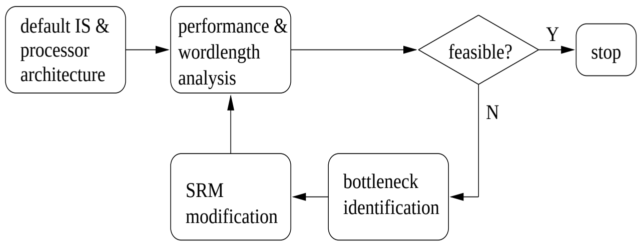

default IS & processor architecture
performance & wordlength analysis
feasible?
Y
stop
N
bottleneck identification
SRM modification

Figure 3.3. Overview of the instruction set design flow

In this optimization flow, depicted in Figure 3.3, we start with a *default instruction set and processor architecture*. Subsequently, the performance of the critical loops is analyzed, as explained in more detail in Chapter 5. If the performance is insufficient, we look for the responsible (virtual) resources in a step called *bottleneck identification*. The SRM is subsequently modified by allocating additional instances of the virtual resources whenever necessary. The essential difference to the HLS flow in Figure 1.3 is that we also consider the instruction width as a criterion in the design process to evaluate the modifications made for the SRM model.

Chapter 4

Construction of an SRM

In this chapter, we present in more detail the method for verifying whether an instruction set has an equivalent SRM outlined in the previous chapter. If it has, the method also gives this SRM.

4.1 Using inequalities for the construction of an SRM

In [34], we presented an approach for deriving an SRM for a given instruction set by making use of inequalities. The basic idea is to generalize the maximum usage of operation types in an instruction set by representing it as a set of inequalities, and then transforming these inequalities into an SRM. The desired set of inequalities is derived from the *operation-type statistics*. An example of these statistics is depicted in the table of Figure 4.1 (b) for the instruction set IS_1 given in Figure 4.1 (a). In the table, rows correspond to the individual instructions listed in Figure 4.1 (a), and columns correspond to (combinations of) operation types. The numbers in the table indicate how many times an operation type (combination) is present in an instruction. For example, the operation type *add*, occurs twice in instruction (1), and the operation types *shift* and *mul* together occur three times in instruction (3). By looking at the largest frequency within each column, the inequalities in Figure 4.1 (c) are derived. Each inequality corresponds to one column. In general, for each combination of operation types $S = \{op_1, \dots, op_n\}$, we have one inequality:

$$N(op_1) + \cdots + N(op_n) \leq \operatorname*{MAX}_{I \in IS}\Big(\sum_{op \in I \cap S} I(op) \Big) \tag{4.1}$$

where $N(op_i)$ is the number of times an operation type appears in an instruction.

The instruction set constraints have now been replaced by a set of inequalities. This set often contains redundancy; in the example of Figure 4.1 (c), inequality (5) can be removed because it is implied by inequality (7). Removing inequalities is advantageous because the complexity of the derived SRM is determined by the number of inequalities. In [34], we provided several rules to remove the redundancy. From the remaining inequalities the SRM is derived, shown in Figure 4.1 (d). For example, inequality (6) is translated to the virtual resource $\{mul, shift\}$, which is abbreviated as MS, with three instances available. For reasons of convenience, we represent a virtual resource by combining the first letters of all those operation types which compose it. An operation type "uses" all the resources from the SRM that it is contained in, e.g., add uses A, AM and AMS at the same time. Note that virtual resource A corresponds to the real functional resource implementing the add operation, while virtual resources AM and AMS have functionalities similar to functional resources but are not real functional resources, explaining the term.

A weakness of this method is that it does not always give an SRM, even if an instruction set has one. Furthermore, a resulting SRM often has redundancies, as the above example illustrates. These redundancies are difficult

(1) [add, add, mul, shift]

(2) [add, add, shift, shift]

(3) [add, mul, mul, shift]

(4) [add, mul, shift, shift]

(a) instruction set IS1

	add	mul	shift	add mul	add shift	shift mul	add shift mul
(1)	2	1	1	3	3	2	4
(2)	2	0	2	2	4	2	4
(3)	1	2	1	3	2	3	4
(4)	1	1	2	2	3	3	4

(b) operation-type statistics

(1) N(add) <= 2

(2) N(mul) <= 2

(3) N(shift) <= 2

(4) N(add) + N(mul) <= 3

(5) N(add)+N(shift) <= 4

(6) N(mul) + N(shift) <= 3

(7) N(add) + N(mul) + N(shift) <= 4

(c) inequalities

#A = 2, #M = 2, #S = 2,

#AM = 3, #MS = 3, #AMS = 4

add -> A, AM, AMS

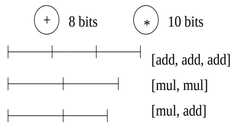
mul -> M, AM, MS, AMS

shift -> S, MS, AMS

(d) static resource model

Figure 4.1. Example of an instruction set with an SRM

to remove. The method is also not well suited for handling complex encodings. Figure 4.2 shows an example where methods from literature [8] and the one above fail to express instruction set restrictions in an SRM-like manner. The difficulty is introduced by the *wordlength constraints* on instructions. There are three adders and two multipliers in the data path. Each add is encoded with 8 bits and each mul with 10 bits. The total wordlength for an instruction is limited to 24 bits, thus all the possible combinations of operation types are: $[add, add, add]$, $[mul, add]$, $[mul, mul]$. The method above will produce the inequalities in Figure 4.2 (b). It contains one redundant inequality: inequality (1). Furthermore inequality (3) allows the combinations of operations such as $[add, mul, mul]$ and $[add, add, mul]$, which are in fact not valid because of the instruction encoding constraints. In contrast, the method explained below generates the inequalities for the instruction set in Figure 4.2 (c), which correctly models the instruction encoding limitation. Using this new inequality the instruction set is verified to have an SRM.



(a) an instruction set

(1) N(add) <= 3

(2) N(mul) <= 2

(3) N(add) + N(mul) <= 3

(b) the incorrect inequalities

(1) N(add) <= 3

(2) N(mul) <= 2

(3) 2 N(add) + 3 N(mul) <= 6

(c) the correct inequalities

Figure 4.2. Example of wordlength constraints

In the next section, we provide a general approach for the construction of SRMs that improves the method of [34]. It solves all the mentioned problems and it gives a valid SRM for the instruction set in Figure 4.2 (a).

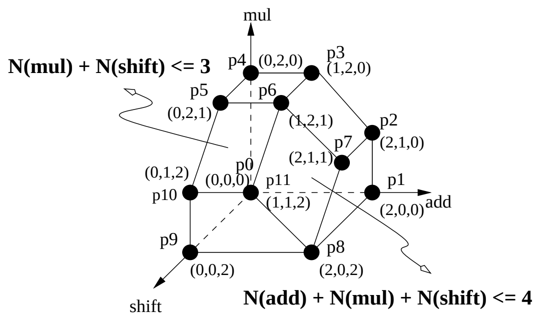
4.2 A general approach for deriving inequalities

In order to find a method that produces a valid SRM for a broader class of instruction sets, we observe the following restriction in the approach of Chapter 4.1: The inequalities do not have weights on the left hand side for each operation type. Assume IS_1 in Figure 4.3 (a) is the full instruction set of a certain VLIW architecture, while IS_2 in Figure 4.3 (b) is the similar but smaller instruction set. The method above will generate the same

SRM for the two instruction sets. However, by adding inequality (6') to the set of inequalities in Figure 4.3 (d), we solve the problem posed in the previous section: Instruction set IS_2 now has a valid SRM, different from the SRM corresponding to IS_1.

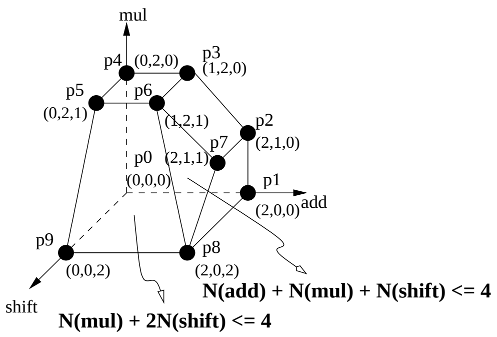
If the operation types are associated with the axes in a multi-dimensional numerical space and the instructions are represented as points in this operation type space geometrically, then the inequalities derived for the instruction set can be perceived as boundaries enclosing those points and the problem is transformed into a *convex hull* problem.

The instruction sets IS_1 and IS_2 and their convex hulls are illustrated in Figure 4.3 (a) and (b). In Figure 4.3 (c), we obtain the same set of inequalities as in Figure 4.1, which in Figure 4.3 (e) leads to the same SRM as in Figure 4.1. Notice that the algorithms used in computational geometry tools, e.g. the cdd [7] package, already omit the redundant inequality (5) in Figure 4.1 (c). Because the last instruction in IS_1 is not valid for IS_2, points $p10$ and $p11$ are not present in Figure 4.3 (b), and the convex hull and the set of inequalities in Figure 4.3 (d) are slightly different from IS_1. Notice that inequality (6'), $N(mul) + 2N(shift) \leq 4$, is generated with the weight on the left hand side automatically. Because of this weight, inequality (3) becomes redundant, which means virtual resource S is not required any more. Also notice that for the same reason, the $shift$ operation uses virtual resource MS twice as is reflected in the SRM of Figure 4.3 (f).



[add, add, mul, shift]
[add, add, shift, shift]
[add, mul, mul, shift]
[add, mul, shift, shift]

(a) instruction set IS1



[add, add, mul, shift]
[add, add, shift, shift]
[add, mul, mul, shift]

(b) instruction set IS2

(1) N(add) <= 2
(2) N(mul) <= 2
(3) N(shift) <= 2
(4) N(add) + N(mul) <= 3

(6) N(mul) + N(shift) <= 3
(7) N(add) + N(mul) + N(shift) <= 4

(c) inequalities for IS1

(1) N(add) <= 2
(2) N(mul) <= 2

(4) N(add) + N(mul) <= 3

(6') N(mul) + 2 N(shift) <= 4
(7) N(add) + N(mul) + N(shift) <= 4

(d) inequalities for IS2

#A = 2, #M = 2, #S = 2
#AM = 3, #MS = 3
#AMS = 4

add -> A, AM, AMS
mul -> M, AM, MS, AMS
shift -> S, MS, AMS

(e) SRM for IS1

#A = 2, #M = 2
#AM = 3, #MS = 4
#AMS = 4

add -> A, AM, AMS
mul -> M, AM, MS, AMS
shift -> MS, MS, AMS

(f) SRM for IS2

Figure 4.3. Examples of SRMs calculated from convex hulls

The general problem of constructing the SRM for an instruction set IS can be formalized as follows: given a set of points (instructions) in the d-dimensional operation type space, determine the convex hull $conv(\text{IS})$ expressed as a set of n linear inequalities.

$$conv(IS) = \{\vec{x} \mid A\vec{x} \leq b\} \tag{4.2}$$

where $d = \mid T \mid$, $A \in \mathcal{R}^{n \times d}$ and $b \in \mathcal{R}^n$.

It remains to determine how to precisely compute (4.2) for a given IS. The basic idea is to use a standard convex hull algorithm applied to the extreme points in the IS. Because we assume prefix closedness of instruction sets, any subinstruction is also a valid instruction. Although a subinstruction is not explicitly listed in the given example instruction sets, it is quite possible to be an extreme point of a convex hull. For example, the four largest instructions in IS_1 can be drawn as points $p7$, $p8$, $p6$ and $p11$ in Figure 4.3 (a). These four points are not sufficient to build $conv(IS_1)$. Point $p1$, for example, represents a subinstruction $[add, add]$, which is obviously an extreme point for this convex hull. In order to construct the complete convex hull, we have to find all the extreme points in the space, which is as complex as the convex hull problem itself. We can simply enumerate all the subinstructions in the instruction set, but this is very inefficient in time. Assuming that the largest instruction can execute m operation types in parallel maximally, and there are k largest instructions in total, then this enumeration complexity is $O(k\ 2^m)$ in the worst case. Obviously, a lot of these points are *redundant* when constructing the convex hull in the sense that they are not extreme points. If they are all contained in the input to the convex hull algorithm, it will increase the runtime of the algorithm although it is polynomial. Unfortunately, removing them is also costly.

As we can see in one example, extreme point $p1$ with coordinates $(2,0,0)$ is the projection of point $p8$ with coordinates $(2,0,2)$ on the plane $shift = 0$. Any point *between* them has no contribution to the convex hull, thus it is redundant. From this intuition we can simplify the enumeration of subinstructions by calculating only the points projected on the planes with dimensions $d-1$, $d-2$, ..., 1 obtained by assuming 1, 2, ..., $d-1$ coordinates to be zero respectively. The complexity of this projection in the worst case is $O(k\ 2^p)$, where p is the number of operation types appearing maximally in one instruction.

The complexity of enumerating only the projected points is usually less than the complexity of complete enumeration of the valid subinstructions (points in the convex hull), because in practice the number of operation types appearing in one instruction is often less than the cardinality m of the largest instruction, especially for instructions with large parallelism. In addition, this number can be minimized by grouping operation types with similar usages, which will further reduce the runtime for the construction of the convex hull.

4.3 Deriving resources

So far, we have explained by means of examples how instruction set constraints can be captured in an SRM based on an approach computing the convex hull of the instruction set. In this section, we give a theorem formulating a necessary and sufficient condition for verifying whether an instruction set has an SRM, thus solving problem 1 as defined in Chapter 3.2. The argument showing that the condition is sufficient includes a transformation from the inequalities describing the convex hull of the instruction set to an SRM. The basic idea of the theorem is that the convex hull of an instruction set defines a new instruction set consisting of all the integer points in the convex hull. If this new instruction set is equal to the original instruction set, then the convex hull provides the basis for an SRM of the original instruction set because it captures precisely all the instruction set constraints. If the new instruction set contains integer points that are not valid instructions in the original set, then the convex hull does not provide an appropriate SRM because the constraints are not sufficiently tight.

Theorem. Let IS be an instruction set. If NIS is the set of all the integer points contained in $conv$(IS), then IS has an SRM if and only if NIS equals IS.

We prove the sufficiency of the condition in this theorem by giving the following SRM. Recall Definition 2 introducing the notion of an SRM and equation (4.2) that describes the convex hull of an instruction set in terms of inequalities. We need to define three aspects:

– the set of virtual resources R of the SRM contains all the sets of operation types corresponding to an inequality in the convex hull description of (4.2);

– each operation type needs w instances of all the virtual resources that it is contained in, where w is the weight of the operation type in the inequality in (4.2) corresponding to the virtual resource;

– the number of instances of a resource equals the bound in the corresponding inequality in (4.2).

To clarify this construction, consider the example instruction set IS_2 of Figure 4.3 (b). Figure 4.3 (d) gives the inequalities describing the convex hull of IS_2. The convex hull consists of five inequalities, which means that the derived SRM has five virtual resources. Inequality (4), for example, corresponds to virtual resource $\{add, mul\}$, as before abbreviated AM. Inequalities (1), (2), (6′) and (7) correspond to virtual resources A, M, MS and AMS, respectively. The number of instances of resource AM is determined by the right hand side of inequality (4), being 3. The other four resources have 2, 2, 4 and 4 instances, respectively. Finally, operation type mul uses one AM resource, because 1 is the weight of mul in inequality (4); it further uses one M resource, one MS resource and one AMS resource. Operation type add uses one A, one AM and one AMS resource; $shift$ uses two MS resources and one AMS resource. Figure 4.3 (f) summarizes the SRM.

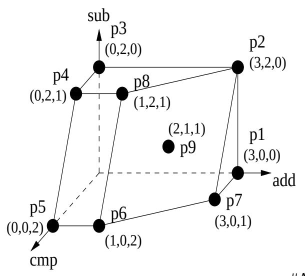
It still needs to be shown that the condition in the theorem that NIS equals IS is sufficient to guarantee that the SRM defined above is an appropriate representation of the instruction set constraints of IS. The construction of the SRM guarantees that a schedule of all the operations in a DFG constrained by the SRM satisfies all the inequalities describing the convex hull of IS. Recall that NIS contains exactly all the integer points satisfying these inequalities. Thus, if NIS equals IS, the inequalities only allow valid instructions which means that any schedule constrained via the above SRM satisfies all instruction set constraints. Thus, IS has an SRM, namely the one provided above.

As an example consider again instruction set IS_2 of Figure 4.3 (b). The inequalities in Figure 4.3 (d) precisely capture all these instructions, i.e., they do not allow any integer solutions that are not instructions in IS_2. Thus, we conclude that IS_2 has an SRM. Figure 4.3 (f) provides an SRM, which is constructed as explained above.

The necessity of the condition that NIS equals IS in the above theorem immediately follows from the following fact: The convex hull of a set of points is the *smallest* convex set containing all these points (Definition 4). In other words, NIS is the smallest set of integer points containing all the instructions in IS that can be described by inequalities of the form given in (4.2). Thus, if NIS contains a point that is not an element of IS, then it is impossible to describe IS via such inequalities. Since there is a one-to-one correspondence between inequalities and SRMs, IS cannot have an SRM.

To illustrate that there are instruction sets without an SRM, consider the example in Figure 4.4. Figure 4.4 gives an instruction set, the inequalities describing its convex hull, and the SRM that can be derived from these inequalities following the above construction. The inequalities allow instructions $[add, add, add, sub, sub]$, $[add, add, add, cmp]$, $[add, sub, sub, cmp]$, $[add, add, sub, cmp]$, $[add, cmp, cmp]$ plus all their subinstructions. Thus, unfortunately, the inequalities allow instruction $[add, add, sub, cmp]$ that is not part of the initial instruction set; it is depicted as point p9 in Figure 4.4 (a). As a result of the above theorem, we may conclude that the initial instruction set has no SRM. Point p9 is part of the plane through points p2, p6, p7 and p8. Thus, it is straightforward to verify that we cannot tighten the corresponding constraint expressed by inequality (4) in such a way that it excludes point p9 but none of the other points that do correspond to valid instructions. Although this is a negative result, it provides us useful information for instruction set design. If we could extend the initial instruction set with instruction $[add, add, sub, cmp]$, the resulting instruction set is guaranteed to have an SRM, which implies all the advantages discussed in this paper.

Note that for certain important classes of instruction sets, it is always possible to derive an SRM using our method. One such class is the class of VLIW instruction sets, which serve as the interface for VLIW architectures



[add, add, add, sub, sub]
[add, add, add, cmp]
[add, sub, sub, cmp]
[add, cmp, cmp]

(a) instruction set

(1) N(add) <= 3
(2) N(sub) <= 2
(3) N(sub) + 2N(cmp) <= 4
(4) N(add) + N(sub) + 2N(cmp) <= 5

(b) inequalities

#A = 3 #S = 2
#SC = 4 #ASC = 5
add -> A, ASC
sub -> S, SC, ASC
cmp -> SC, SC, ASC, ASC

(c) static resource model

Figure 4.4. Verifying an instruction set having no SRM

that are used quite often in media processing. Our method is a generalization of the methods presented in [8], [29], and [34].

Chapter 5

Design of the instruction set architecture from an SRM

Now that we have explained the SRM model, we are going to use that model to do instruction set 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 cannot 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.

5.1 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 next.

When applying software pipelining techniques to loop kernels, the *initiation interval (II)* is an important criterion for measuring the performance. An II is the period between the start times of the execution of two successive loop-body iterations. The *minimum initiation interval (MII)* is the lower bound of the II. The MII can be determined either by a critical resource that is fully utilized or a critical chain of dependencies running through the loop iterations. One lower bound, the *resource constrained MII (ResMII)*, is derived by calculating, in total, the usage requirements for each resource imposed by one iteration of the loop.

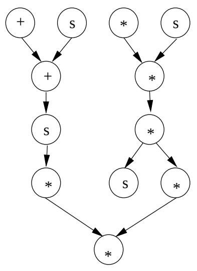
Suppose a loop containing 14 add operations is mapped on a data path containing three adders and each adder takes one clock cycle to execute. Then we need at least $\lceil \frac{14}{3} \rceil = 5$ clock cycles to execute the loop iteratively. By doing this calculation for every available resource, we obtain the lower bound ResMII on the initiation interval II 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 lower bound estimate can therefore be used for bottleneck identification.

In case of instruction set constraints and the corresponding SRM, virtual resources should be taken into account for the performance estimation. The ResMII is no longer simply totaling of resource usages for each resource because some operations are mapped into more than one instance of virtual resources. Thus the ResMII has to be updated with respect to the multiple usage of the virtual resources [35]. Suppose a virtual resource M with b number of instances available, and M is used by a set of operation types $S = \{op_i, i = 1, \cdots, n\}$. Also assuming that the number of operations mapped to operation type op_i is n_i, then ResMII is updated as:

$$\text{ResMII} = \underset{M \in R}{\text{MAX}}\left(\frac{\sum_{i=1}^{n} a_i n_i}{b}\right) \tag{5.1}$$

where R is the set of virtual resources and a_i is the i-th element in vector A in equation (4.2).

For example, for the instruction set in Figure 4.3 (b), inequality (6') in Figure 4.3 (d) indicates that the $shift$ operation uses two instances of the resource MS, of which four are available. For a DFG in Figure 5.1 (a), the number of operations using each operation type add, mul and $shift$ can be expressed as $n_{add} = 2$, $n_{mul} = 6$, $n_{shift} = 4$ respectively. Assume functional resources $adder$, $multiplier$ and $shifter$ are available, each with two instances. Thus the estimated lower bound for the initiation interval with respect to the functional resources in the data path is 3, shown in Figure 5.1 (b). While the DFG in Figure 5.1 (a) is executed by instruction set IS_2, the corresponding virtual resources in Figure 4.3 (d) modify the estimation as depicted in Figure 5.1 (c), which is one clock cycle longer than the original estimation based on teh functional resources. This lower bound is very tight; we can apply the loop folding technique directly by first mapping the operations to the virtual resources, as depicted in Figure 4.3 (f). Figure 5.2 shows the optimal scheduling result and the virtual resource usages in the loop kernel.



(a) A data flow graph

$n_{add} = 2 \quad n_{mul} = 6 \quad n_{shift} = 4$

$$\text{MII} = \max\left(\frac{n_{add}}{\#A}, \frac{n_{mul}}{\#M}, \frac{n_{shift}}{\#S}\right)$$
$$= \max(2/2, 6/2, 4/2)$$
$$= 3$$

(b) ResMII estimated with functional resources

$$\text{MII} = \max\left(\frac{n_{add}}{\#A}, \frac{n_{mul}}{\#M}, \frac{n_{add}+n_{mul}}{\#AM}, \frac{n_{mul}+2n_{shift}}{\#MS}, \frac{n_{add}+n_{mul}+n_{shift}}{\#AMS}\right)$$
$$= \max(2/2, 6/2, (2+6)/3, (6+2*4)/4, (2+6+4)/4)$$
$$= 4$$

(c) ResMII estimated with virtual resources

Figure 5.1. DFG and the MII estimation

Like in the bottleneck analysis of high-level synthesis of ASICs in Figure 1.3, the bottleneck of performance now can be identified from the updated lower bound estimation and can be relieved by allocating additional (virtual) resources. 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, because they describe both the resource availability and, for each operation type, the usage of that resource. The way that we relieve the bottleneck is explained in the following section.

5.2 Modification of the SRM

We consider two possibilities to modify the SRM [35].

- 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 most impact on the lower bound.

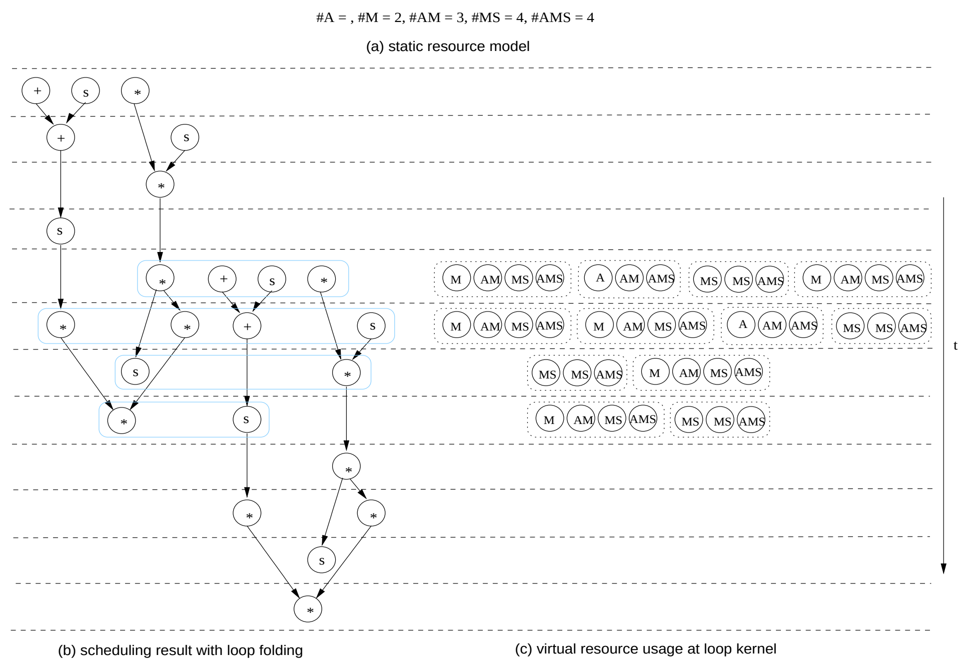

#A = , #M = 2, #AM = 3, #MS = 4, #AMS = 4
(a) static resource model
t
(b) scheduling result with loop folding
(c) virtual resource usage at loop kernel

Figure 5.2. Scheduling result with loop folding and the virtual resource usage in the kernel

Table 5.1. Modification of the SRM

	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	26 bits	28 bits

An example in Table 5.1 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 4.2. The performance requirement of the loop under consideration is given as $\mathrm{II} = 5$.

In Table 5.1, the first column corresponds to the initial design depicted in Figure 4.2 (c). The 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 second 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 SRM modification, II is now lower bounded by 6. A new instruction $[add, add, mul]$ is added to the instruction set, thereby increasing the instruction width to 26 bits. In the third 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 that the design in the third column meets the performance requirements, although it increases the wordlength to 28 bits. We consider a more elaborate example in the next chapter.

Chapter 6

Experimental results

In this chapter, we provide several experiments of constructing the SRM for certain instruction set and applying the constructed SRM to an existing scheduler. These experiments are automated when all the maximum instructions are provided. We also provided a small case study of instruction set design, which is performed manually. The first example shows that our method can deal with very flexible designs using complex instructions in DSPs and ASIPs. The second experiment proves that the SRM approach can be used to evaluate the restrictiveness of an instruction set. This provides quite useful information for the instruction set design. Loop folding can be applied efficiently to loop kernels with the SRM approach, which is presented by the third example.

6.1 Complex instructions in TMS320C25

Often in an instruction set for a VLIW architecture, operation types are represented by a reservation table and are constrained by issue slots. In this case, approaches in [8] [29] can translate the issue slot constraints to static resource constraints and apply the results for a scheduler. However, the instruction sets of DSPs or ASIPs are designed with many irregularities. For instance, in TMS320C25 processor, complex instructions $[lt, pac]$, $[mpy, pac]$, $[lt, apac]$ are introduced to exploit the limited parallelism in the data path. Such complex instruction constraints cannot be expressed as a reservation table, because not all the combinations of the operation types are valid, e.g., $[mpy, apac]$ is not a valid instruction. Thus virtual resources cannot be derived directly using the reservation table approaches, but our method can still deal with it.

[lt]
[mpy]
[pac]
[apac]
[sacl]
[lt, pac]
[lt, apac]
[mpy, apac]

(a) (complex) instructions

#SLM = 1, #SMP = 1, #SPA = 1 #SLMPA = 2

lt -> SLM, SLMPA
mpy -> SLM, SMP, SLMPA
pac -> SMP, SPA, SLMPA
apac -> SPA, SLMPA
sacl -> SLM, SMP, SPA, SLMPA

(b) SRM for this instruction set

Figure 6.1. Instructions for TMS320C25 and the SRM

Instructions for the TMS320C25 processor are shown in Figure 6.1(a). They indicate potential parallelism to be exploited. This instruction set can be proven to have an SRM and it is given in Figure 6.1(b). We have implemented UTDSP benchmarks [30] for the TMS320C25 processor and the result is in Table 6.1.

Table 6.1. Scheduling result of UTDSP benchmarks on TMS320C25 processor

	without SRM				with SRM			
	|V|	|E|	#instruction	T(ms)	|V|	|E|	#instruction	T(ms)
fft	24	26	24	60	68	26	20	920
fir filter	5	8	5	0	12	8	5	10
iir filter	17	22	17	40	42	22	14	350
lattice filter	14	17	14	10	29	17	11	60
lms fir filter	7	10	7	10	18	10	6	10

The column "without SRM" assumes there is no approach to exploit the instruction-level parallelism (ILP) in the processor, while "with SRM" reports the scheduling results by constructing the SRM and applying the SRM to the scheduler. In each case, the first and second column show the number of nodes and edges in the DFG of each example, while the third and fourth column report the number of instructions and runtime of the scheduler. Since the complex instructions are quite small, the runtime of constructing the SRM can be ignored. In the first case, each operation is encoded with one specific instruction, after scheduling the number of clock cycles equals the total number of operations. While in the second case, since each operation can use multiple virtual resources, during the implementation, we explicitly expressed the resource usage of each operation, which causes the increase of the number of nodes. The number of instructions is reduced compared to the first case for all the examples except for 'fir filter'. The runtime is increased for each example; it is dependent on the usage of virtual resources for each operation.

6.2 Evaluating the restrictiveness of an instruction set

One of the merits of the SRM approach is that by performing instruction scheduling for applications with different SRMs, one can directly see from the results how restrictive an IS is. Experiments below perform the scheduling and register binding for benchmarks with two instruction sets IS_1 and IS_2 in Chapter 4. Assume that the hardware resources are ALU, MUL and $SHIFT$, each with two number of instances available. Each resource has a delay of one clock cycle. Table 6.2 lists the results for 'fdct', 'idct', 'ar' for AR filter and 'wdf' for fifth-order digital elliptical wave filter. The second and third columns show the latency and register requirements (one register file for allocating all the values) assuming that the all the given functional resources can be used in parallel maximally, the fourth and fifth columns for IS_1 and the sixth and seventh columns for IS_2. #in is the number of inequalities of the SRM.

Table 6.2. Scheduling and register binding results for benchmarks

	FS		IS1		IS2	
	L	RF	L SRM	RFSRM	L SRM	RFSRM
fdct	13	12	16	12	21	14
idct	14	11	15	11	20	13
ar	10	6	11	7	14	8
wdf	16	8	16	8	19	11
#in	3		6		4	

From this table we can see that both the latency and register requirements when the functional resources can be used maximally are minimum for all the examples. Although using the same resources, IS_2 is obviously more restrictive than IS_1, since all the scheduling results are increased quite substantially. Register requirements are also increased because when some operations are postponed during scheduling in order to meet the instruction set constraints, the values they consume have to be stored in registers for a longer time, which increases the register pressure.

Table 6.3. TMS320C62x instruction set

	L	S	D	M
Integer Adder	add	add	add	
	sub	sub	sub	
	mov	mov	mov	
	cmpgt			
	cmplt			
Logic	and	and		
	or	or		
	not	not		
Shift		shl		
		shr		
Load Store			ld	
			st	
Multiplier				mpy

6.3 Loop folding with the SRM approach

VLIW instruction sets have an SRM because of their orthogonal instruction format, such as the TMS320C62x architecture. The C62x has two identical data paths with four functional units each. Each data path has 16 32-bit registers. Table 6.3 shows part of the instructions for one group of data path supported by TMS320C62x.

Using the convex hull approach, virtual resources can be created for this instruction set. SRMs can also be applied to software pipelining easily, since it only provides a static boundary without modifying the scheduling and register binding algorithms themselves. Table 6.4 shows the scheduling and register binding results for GSM speech-coding algorithms. 'Convolution' and 'viterbi' are the half-rate convolutional encoder and one_viterbi_step for viterbi decoder in channel codec. 'weight' and 'inverse' are the weighting_filter and APCM_inverse_quantization algorithms for regular pulse excitation encoding in speech codec. 'reflect' is the reflection_coefficients for lpc code.

Table 6.4. Scheduling and register binding results for GSM speech-coding algorithms

	|V|	II	RF	|V|'	T(ms)
invers	6	2	5	12	10
convol	14	5	8	22	20
reflect	19	7	6	34	170
weight	39	11	7	57	640
viterbi	55	19	10	108	15570

In Table 6.4, $|V|$ is the number of operations in each application. II is the minimal initiation interval when software pipelining is applied. RF is the register requirements under the minimal initiation interval restrictions. The fifth column reports the number of nodes after mapping operations to virtual resources. The last column reports the total runtime including mapping, scheduling and register binding.

6.4 Instruction set design

We also performed a case study for the instruction set design using the SRM approach.

[a, m, m]
[a, s, s, m]
[a, a, s, l]
[a, a, s, m]
[a, a, a, m]
[a, a, a, s, s]

(a) instruction set

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

(b) SRM and the estimated initiation intervals

Figure 6.2. An instruction set and the SRM

For the instructions given in Figure 6.2 (a) an SRM can be obtained using the approach in Chapter 4. We use the fast method for performance evaluation explained in Chapter 5 rather than performing detailed scheduling. Since the topology of the DFG of the loop is irrelevant for this analysis, we give 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. From the column, we obtain that the lower bound MII is 8. Assuming the objective II is set as 6, this design is far below the performance requirements.

Table 6.5. Modification for candidates (3) and (4)

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

Table 6.5 shows the different results by applying the modification methods in Chapter 5 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 6.2 (a). The fifth column calculates the wordlength with the new instruction set.

Table 6.6. Modification for candidates (5) and (6)

	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

From Table 6.5 we can see that all designs sufficiently reduce the bottleneck. We choose the design of row three for the 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 6.2 and the same procedure is repeated and shown in Table 6.6. From this table, 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.

This example shows that the SRM approach provides the good bases for tuning between performance and code size during the instruction set design. This administrative approach is quite efficient for current embedded processor design because without looking at the detailed encoding and performing exact scheduling for the applications the designer can already obtain the knowledge of which adjustment he should take.

Chapter 7

Conclusions

In this paper, we propose a method for code generation for highly encoded instruction set processors and instruction set design. The purpose of the method is to deal with the strong phase coupling between instruction selection and scheduling, caused by the instruction set. We replace the instruction set constraints with a set of virtual resources that implement the individual operations contained in the instruction set. This reduces the need for explicit instruction selection, because any schedule satisfying the resulting static resource constraints can be implemented by the instruction set. With this static resource model, the scheduler has the opportunity to generate better schedules in terms of timing and register requirements. Furthermore, software pipelining can be applied more effectively to exploit the available ILP.

Our experiments demonstrate that a static resource model of an instruction set offers the possibility to measure the effectiveness of an instruction set with respect to exploiting the processor resources. This enables instruction set designers to make tradeoffs between the performance and the code size associated with an instruction set.

The SRM model also allows instruction set design in terms of allocating (virtual) functional resources, a practical method used in the 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.

Chapter 8

Future work

The key of the SRM approach is to represent some architectural constraints, e.g. encoding constraints with static resource constraints, i.e., resources only provide a fixed upper limit and any usage within this limit is valid. It is natural to extend this work to support other architectural constraints, e.g. interconnection constraints and bus constraints. In order to avoid using a large register file because of the large area and long access time, designers tend to partition the architecture into clustered data paths. As a consequence, explicit move operations have to be inserted between different clusters for the communication. In order to make use of the register files in a flexible way and reduce explicit move operations as much as possible, some registers are maintained as a common part, which can be written by all the functional units, while others are clustered for access of local functional units, which we think can be modeled by an SRM. Another application of SRM is to support SIMD instruction sets with some improvement of register allocation, since multiple values have to be stored in one register when multiple operations are executed at the same clock cycle by one instruction.

In high-level synthesis, multiple-precision specifications used to be implemented on hardware in a simple way, i.e., operations are mapped on wider functional resources, filling the operands with 0's and discarding some bits of the solutions produced. Some work [10] suggests the implementation in more efficient way, e.g. the execution of an operation during several cycles or the execution of multiple operations on a unique resource during the same cycle. Also this resource selection and binding has to be integrated with scheduling to obtain high performance. Efficient results can be obtained using the SRM approach for multiple-precision systems because the data path precisions resemble instruction set constraints very closely.

Bibliography

[1] Avis D. and K. Fukuda, *A pivoting algorithm for convex hulls and vertex enumeration of arrangements and polyhedra*. Discrete and computational geometry, vol. 8 (1992), p. 295-313.

[2] Avis D. and D. Bremner, R. Seidel, *How good are convex hull algorithms?*. Computational geometry: theory and applications, vol. 7 (1997), p. 265-301.

[3] Aho A.V. and S.C. Johnson, *Optimal code generation for expression trees*. Journal of ACM, vol. 23 (1976), no. 3, p. 488-501.

[4] Aho A.V. et al., *Code generation using tree matching and dynamic programming*. ACM transactions on programming languages and systems, vol. 11 (1989), no. 4, p. 491-516.

[5] Araujo G. and S. Marlik, M. Lee, *Using register-transfer paths in code generation for heterogeneous memory-register architectures*. In: Proc. 33rd Design automation conference, Las Vegas, Nevada, USA, June 3-7, 1996. New York: ACM, 1996. P. 591-596.

[6] Bashford S. and R. Leupers, *Phased-coupled mapping of data flow graphs to irregular data paths*. Design automation for embedded systems, vol. 4 (1999), no. 2/3, p. 119-165.

[7] ftp://ftp.ifor.math.ethz.ch/pub/fukuda/cdd/ (Last access: April 2002.)

[8] Eisenbeis C. and Z. Chamski, E. Rohou, *Flexible issue slot assignment for VLIW architectures*. Presentation at the 4th Int. workshop on software and compilers for embedded systems, St. Goar, Germany, September 1-3, 1999. Available through the first author (affiliation: INRIA, Rocquencourt, France).

[9] Emmelmann H. et al., *Beg - a generator for efficient back ends*. SIGPLAN notices, vol. 24 (1989), no. 7, p. 227-237.

[10] Ercegovac M. and D. Kirovski, M. Potkonjak, *Low-power behavioral synthesis optimization using multiple precision arithmetic*. In: Proc. 36th Design automation conference, New Orleans, LA, USA, June 21-15, 1999. New York: ACM, 1999. P. 568-573.

[11] Fauth A. et al., *Describing instruction set processors using nML*. In: Proc. European design and test conference, Paris, France, March 6-9, 1995. Los Alamitos: IEEE computer society, 1995. P. 503-507.

[12] Fraser A.W. et al., *Engineering a simple, efficient code-generator generator*. ACM letters on programming languages and Systems, vol. 1 (1993), no. 3, p. 213-226.

[13] ART Designer Tutorial, Frontier Design, Melbourne, Florida, USA, 2001. V2.2 rev25. Tutorial.

[14] Hanono S. and S. Devadas, *Instruction selection, resource allocation and scheduling in the AVIV retargetable code generator*. In: Proc. 35th Design automation conference, Anaheim, CA, USA, June 9-13, 1998. New York: IEEE, 1998. P. 510-515.

[15] Liem C. and T. May, P. Paulin, *Instruction-set matching and selection for DSP and ASIP code generation*. In: Proc. European design and test conference, Grenoble, France, September 19-23, 1994. Los Alamitos: IEEE computer society, 1994. P. 31-37.

[16] Leupers R. and P. Marwedel, *Instruction selection for embedded DSPs with complex instructions*. In: Proc. European design automation conference, Paris, France, March 11-14, 1996. Los Alamitos: IEEE computer society, 1996. P. 200-205.

[17] Leupers R., *Retargetable code generation for digital signal processors*. Dordrecht (The Netherlands): Kluwer, 1997.

[18] Leupers R., *Code selection for media processors with SIMD instructions*. In: Proc. Design automation and test in Europe conference and exhibition, Paris, France, March 27-30, 2000. Los Alamitos: IEEE computer society, 2000. P. 4-8.

[19] Leupers R., *Register allocation for common subexpressions in DSP data paths*. In: Proc. Asia south pacific design automation conference, Pacifico Yokohama, Japan, January 25-28, 2000. Piscataway: IEEE, 2000. P. 235-240.

[20] Mesman B. and C. Alba-Pinto, C. van Eijk, *Efficient scheduling of DSP code on processors with distributed register files*. In: Proc. International symposium on system synthesis, Boca Raton, Florida, USA, November 1-4, 1999. Los Alamitos: IEEE computer society, 1999. P. 100-106.

[21] McMullen P. and G.C. Shephard, *Convex polytopes and the upper bound conjecture*. Cambridge (UK): Cambridge university press, 1971.

[22] Novack S. and A. Nicolau, N. Dutt, *A unified code generation approach using mutation scheduling*. In: Marwedel P. and G. Goossens (eds), *Code generation for embedded processors*. Dordrecht (The Netherlands): Kluwer, 1995. Chapter 12.

[23] Paulin P. et al., *FlexWare: a flexible firmware development environment for embedded systems*. In: Marwedel P. and G. Goossens (eds), *Code generation for embedded processors*. Dordrecht (The Netherlands): Kluwer, 1995. Chapter 4.

[24] Paulin P. and C. Liem, *Embedded systems: tools and trends*. Tutorial at the European design and test conference, Paris, France, March 11-14, 1996. Available through the first author (affiliation: STMicroelectronics, Central R&D, Ottawa, Canada).

[25] Preparata R.P. and M.I. Shamos, *Computational geometry: an introduction*. Heidelberg (Germany): Springer, 1985.

[26] Rau B.R. and C.D. Glaeser, *Some scheduling techniques and an easily schedulable horizontal architecture for high performance scientific computing*. In: Proc. 14th Workshop on microprogramming, Chatham, Massachusetts, USA, October 12-15. New York: IEEE, 1981. P. 183-198.

[27] Xtensa, Tensilica Inc., http://www.tensilica.com/technology.html. (Last access: April 2002.)

[28] Texas Instruments, *TMS320C6000 CPU and instruction set reference guide*, 2000. http://dspvillage.ti.com/. (Last access: April 2002.)

[29] Timmer A. and M. Strik, J. van Meerbergen, J. Jess, *Efficient code generation for in-house DSP cores*. In: Proc. European design and test conference, Paris, France, March 6-9, 1995. Los Alamitos: IEEE computer society, 1995. P. 244-249.

[30] UTDSP Benchmark Suite. http://www.eecg.toronto.edu/˜corinna/DSP/infrastructure/UTDSP.html (Last access: April 2002.)

[31] van Praet J. and G. Goossens, D. Lanner, H. De Man, *Instruction set definition and instruction selection for ASIPs*. In: Proc. International symposium on high-level synthesis, Niagara-on-the-lake, Ontario, Canada, May 18-20, 1994. Los Alamitos: IEEE computer society, 1994. P. 70-75.

[32] Wilson T. and G. Grewal, B. Halley, D. Banerji, *An integrated approach to retargetable code generation*. In: Proc. International symposium on high-level synthesis, Niagara-on-the-lake, Ontario, Canada, May 18-20, 1994. Los Alamitos: IEEE computer society, 1994. P. 11-16.

[33] Woudsma R. et al., *EPICS: A flexible approach to embedded DSP cores*. In: Proc. 5th Int. conference on signal processing, applications and technology, Dallas, Texas, USA, October 18-21, 1994. Waltham (MA, USA): DSP associates, 1994. P. 506-511.

[34] Zhao Q. and T. Basten, B. Mesman, C.A.J. van Eijk, J.A.G. Jess, *Static resource models of instruction sets*. In: Proc. Int. symposium on system synthesis, Montreal, Quebec, Canada, September 30-October 3, 2001. New York: ACM, 2001. P. 159-164.

[35] Zhao Q. and B. Mesman, T. Basten, *Practical instruction set design and compiler retargetability using static resource models*. In: Proc. Design, automation and test in Europe, Paris, France, March 4-8, 2002. Los Alamitos: IEEE computer society, 2002. P. 1021-1026.