A Graph Based Processor Model for Retargetable Code Generation

J. Van Praet D. Lanneer G. Goossens W. Geurts H. De Man

IMEC, kapeldreef 75, 3001 Leuven, Belgium
e-mail : vanpraet@imec.be

Abstract

Embedded processors in electronic systems typically are tuned to a few applications. Development of processor specific compilers is prohibitively expensive and as a result such compilers, if existing, yield code of an unacceptable quality. To improve this code quality, we developed a retargetable and optimising code generator. It uses a graph based processor model that captures the connectivity, the parallelism and all architectural peculiarities of an embedded processor. In this paper, the processor model is presented and we formally define the code generation task, including code selection, register allocation and scheduling, in terms of this model.

1 Introduction

Designers of electronic systems — such as consumer electronics, communication systems and multi-media related products — more and more incorporate programmable processors in their systems. Programmability offers them cost-effective hardware reuse and the flexibility to support last minute specification changes or to add new features to the system. The requirements of programmability, low cost, and low power have resulted in a new class of application specific instruction set processors (ASIPs). These are a hybrid form of custom architectures and standard processors, offering an instruction set and hardware implementation that are optimised for a small number of applications.

A code generator and an instruction set simulator are the key tools to aid a designer when developing software. In the context of cost-effective ASIPs, the optimality of the generated code is very important. Surveys indicate however that the quality of code generated for embedded processors is still problematic, even for general purpose fixed point DSP processors [10]. Moreover, because of the small number of applications to be mapped onto a new ASIP, only a small effort can be spent to develop these supporting tools.

To meet the above requirements, we developed the CHESS *retargetable* code generation and simulation environment. The basis of this environment is a *processor model*, called instruction set graph (ISG). All tools are processor independent and are retargetable by only providing a new or modified ISG processor model.

This paper describes the ISG processor model and its use. Section 2 explains the requirements for an ASIP processor model in further detail and compares with related work. Section 3 provides the basics of the ISG model. Section 4 shows how the code generation tasks can be defined formally in terms of the ISG model. Finally, Section 5 and Section 6 conclude with some comments about the implementation and results of the CHESS code generator, and summarise current and future work.

2 Problem definition and related work

Our CHESS code generator targets fixed-point application-specific and general-purpose DSP processors. In this paper, we will use the Analog Devices ADSP-2111 processor [2] to illustrate the concepts of our model. Part of its data path — we left out the address generation units — is shown in Figure 1.

Heterogeneous register structure. Like most other fixed-point DSP processors, the ADSP-2111 has a heterogeneous register structure, with distributed register(file)s that have a restricted connectivity. Functional units can only write to and get their operands from a limited set of

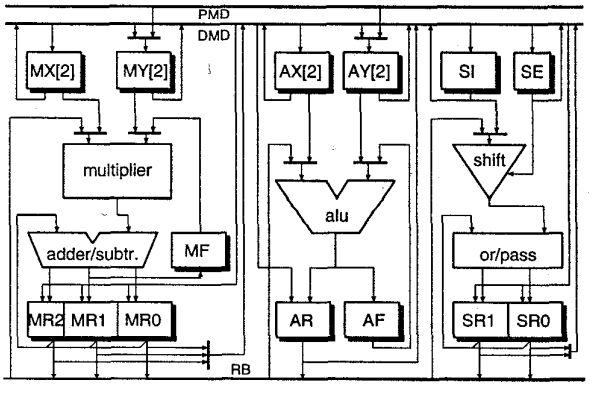

Figure 1: Data path of the ADSP-2111 processor.

registers. For example, the left input of the alu can be AX (2 fields), AR, SR0, SR1, MR0, MR1 or MR2; the right input AY (2 fields) or AF, and its output can be AR or AF.

Classically, in a compiler, processor connectivity is expressed by enumerating all executable operation patterns that end in registers or memory. For (load-store) architectures with a homogeneous register structure, i.e. with one central register file, code selection is independent from the allocation of variables to the register file. For a processor with a heterogeneous register structure, the connectivity of patterns to different register(file)s makes the pattern base very large. For example, for each alu operation, the pattern base contains a pattern for every supported combination of possible input and output registers. This makes code selection and register allocation two interdependent tasks. Wess [11] and Araujo [3] have integrated code selection and register allocation for heterogeneous register structures, extending the technique of tree covering based on dynamic programming that is described in [1].

To reduce the pattern base, Liem [7] groups registers into overlapping classes, according to their connections, and uses these in the patterns. Register allocation is performed separately using a modified left-edge algorithm. Still, all above approaches need a rather big pattern base, except for Nowak's [8], who used a graph as processor model. This graph model captured connectivity and instruction encoding, but no hardware conflicts. Our model also models these hardware conflicts.

Parallelism. Most DSP processors can fetch one or two operands, update memory pointers and perform an arithmetic operation in parallel. Although the data path of the ADSP-2111 in Figure 1 suggests that some arithmetic units could work in parallel, this is not true because of instruction encoding. But not only encoding restricts parallelism, sometimes parallelism available in the instruction set is restricted by the hardware. Consider a memory fetch to be stored in register AR in parallel with an alu operation which normally also stores its result in the AR register. Although it has a legal encoding, this is an "illegal" instruction, because of a hardware conflict on the AR register.

In classical compilers the actual code generation phases, i.e. code selection and register allocation, do not consider parallelism; it is not even contained in their processor models. They generate non-parallel (vertical) code with as a cost the number of vertical instructions. A post-processing pass then compacts these instructions into parallel (horizontal) instructions. However, the resulting code quality largely depends on which vertical instructions are generated, as some of them can be executed in parallel and others can not. Parallelism should thus also be considered in code selection and register allocation. Hence it must be supported by the processor model.

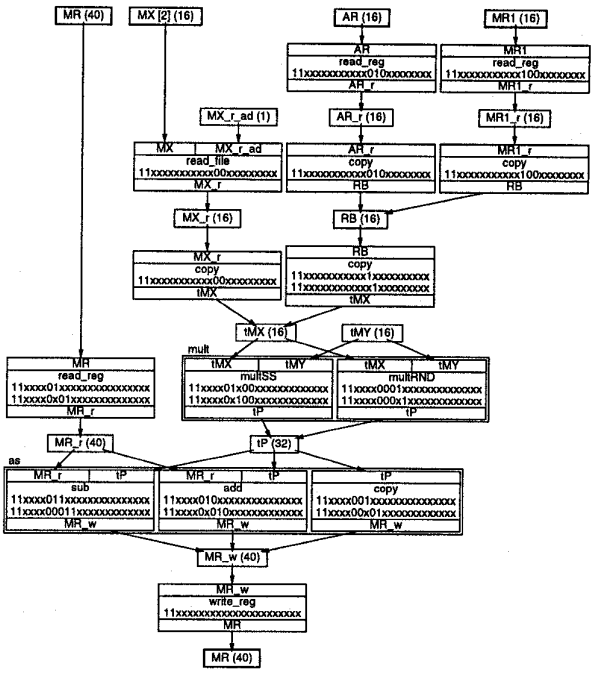

Figure 2: Partial ISG for the ADSP-2111.

3 A bipartite graph as a processor model

The instruction set graph (ISG) is a directed bipartite graph $G_{ISG}\langle V_{ISG}, E_{ISG}\rangle$ with $V_{ISG} = V_S \cup V_I$, where V_S contains vertices representing storage elements in the processor and V_I contains vertices representing its operations. The edges in $E_{ISG} \subset (V_S \times V_I) \cup (V_I \times V_S)$ represent the connectivity of the processor and model data flow from storage, through ISG operations, to storage.

Figure 2 contains a small part of the ISG for the ADSP-2111, the details of which will be explained below.

ISG operations. ISG operations are primitive processor activities transforming values in storage elements into other values in other storage elements.

In each instruction, the processor executes a number of ISG operations. Conversely, a certain ISG operation can be enabled by several instructions. The set of instructions that enables an operation i in the ISG is called its *enabling condition* and denoted by enabling(i). We currently assume that the binary encoding of instructions is available, and use it in the enabling conditions. However, the enabling conditions could also be defined in a more abstract way, using assembler-like symbols for instruction parts. In Figure 2, the enabling conditions are shown in a binary cubic representation, with "x" meaning "don't care". A subset of ISG operations $V_{I_o} \subset V_I$ is said to have an *encoding conflict* when $\bigcap_{i \in V_{I_o}} \text{enabling}(i) = \emptyset$.

A multiplexer is for example modelled as a set of copy operations having a common output storage element such that all pairs of copy operations have encoding conflicts.

Storage. As in [5], we distinguish between two kinds of storage elements :

1. *Static* storage holds its value until explicitly overwritten and can be read several times. A static storage element has a certain capacity of values it can contain at the same time (e.g. the number of fields in a register file). Static storage consists of memory and controllable registers, respectively denoted by the sets V_M and V_R.

2. *Transitory* storage passes a value from input to output with a certain delay. A transitory[1] can only contain *one* value at a time. Examples are buses and wires, which have zero delay, and pipeline registers which have a non-zero delay. Transitories form the set V_T.

Together, the storage elements define a *structural skeleton* of the target machine ($V_S = V_M \cup V_R \cup V_T$). In Figure 2, storage elements are depicted as small rectangular boxes, each having a label denoting their bit-width between parentheses. Those at the top and bottom are registers, all others are transitories.

An interesting feature of transitories is that they are used to model *hardware conflicts* (or structural hazards) in code generation. A hardware conflict is represented as an access conflict on a transitory. The code generator will avoid access conflicts on transitories, by allowing at most one operation to write to each transitory in each machine cycle. As an example, the code generator will never enable different tristate drivers, modelled as copy operations, to write to the same bus, modelled as a transitory, even if they have non-exclusive enabling conditions. Another example is the case of read/write ports of static storage, which are modelled as transitories to make the code generator check for *port conflicts*.

In summary, memory and register nodes are included in the ISG as they are present in the architecture. Transitories on the other hand, are not necessarily uniquely related to physical interconnect resources in the architecture. An operation may for example encapsulate a physical interconnection, or a transitory may be needed to connect the parts of an operation that is artificially split. It is however crucial that the correct processor behaviour is represented, including the hardware conflicts.

4 Code generation using the ISG model

In this section we will explain how we use the ISG processor model to generate code for the execution of a given algorithm on a given processor. The algorithm is given as a data flow graph (DFG) which also takes the form of a bipartite graph $G_{DFG}\langle V_{DFG}, E_{DFG}\rangle$, where $V_{DFG} = V_O \cup V_V$ with V_O representing the operations and V_V representing the values they produce and consume. The edges in $E_{DFG} \subset (V_O \times V_V) \cup (V_V \times V_O)$ represent the data flow. Code generation then consists in finding a mapping of $G_{DFG}\langle V_{DFG}, E_{DFG}\rangle$ onto $G_{ISG}\langle V_{ISG}, E_{ISG}\rangle$ with values in V_V mapped on storage elements in V_S and the DFG operations of V_O on corresponding ISG operations of V_I.

The code generation task is split in subsequent phases. First, during the code selection phase, we decide on which values will be bound to transitories [9]. In the remainder of this paper, we will assume that all transitories have zero delay and that each operation executes within a single cycle. A data dependency of which the corresponding value is bound to a transitory then implies that the involved DFG operations will be executed during the same cycle. Therefore these operations may not be conflicting, otherwise the value must be bound to a static storage element. Conflict free DFG operations that are to be executed in the same cycle because of the binding of their data dependencies, are grouped, and each implementation of such a group is called a *bundle* [5]. After this, during the register allocation phase, the remaining values are bound to static storage elements and the DFG is completed with the necessary data transfers. Finally, during the scheduling phase, the bundles are bound to time. Below, we give more details on how the DFG is bound to the ISG.

Refinement. To represent the different ways in which a DFG operation can be mapped on the ISG, an operation type hierarchy is used. Figure 3 shows an example of this hierarchy for a subtraction, where four implementations exist for the abstract sub operation type. Each implementation is a subtype of an abstract operation type, e.g. subXY is a subtype of sub. The abstract operations and the operations in the ISG form a library $\mathcal{L}$ of operation types for the DFG; each DFG operation is an instance of an operation in $\mathcal{L}$: $\forall o \in V_O, \exists l \in \mathcal{L} : \text{type}(o) = l$.

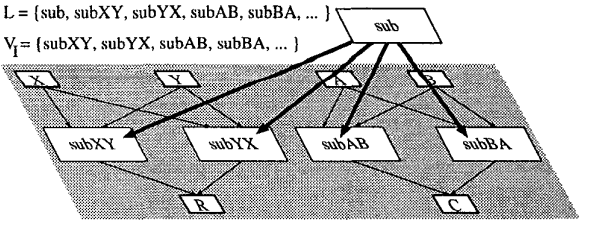

Figure 3: Example of the type hierarchy in the operation type library $\mathcal{L}$. The gray plane represents the ISG.

[1] A transitory storage element will simply be called a transitory.

During code generation, DFG operations are refined until they are instances of ISG operations, so they can be executed by the processor. Each operation $o \in V_O$ with $\text{type}(o) \in \mathcal{L}$ is replaced by a refined operation r such that :

$$\text{type}(r) = i \wedge i \in V_I \wedge i \text{ is a subtype of } \text{type}(o) \quad (1)$$

Binding data dependencies. Consider a data dependency between two refined DFG operations r_1 and r_2, with corresponding value $v_1 \in V_V$. Assume that the DFG operations are bound by the code generator to the ISG operations $i_1 = \text{type}(r_1)$ and $i_2 = \text{type}(r_2)$. We define the function $\text{output}(i, n)$ that returns the storage element on which i writes its n-th output and the function $\text{input}(i, n)$ that returns the storage element from which i reads its n-th input. To simplify the notation, we assume in the following that the respective integers n are chosen according to the data dependency under consideration.

The code generator has different alternatives to bind a data dependency to the ISG. If a data dependency is bound to a path in the ISG that does not include other storage elements than zero-delay transitories, the involved DFG operations will be executed during the same cycle. We call this a *direct* data dependency. Figure 4(a) shows the binding of a data dependency where $\text{output}(i_1, n) = \text{input}(i_2, n') = t$ with $t \in V_T$. Value v_1 is then bound to the transitory t, denoted $\text{carrier}(v_1) = t$. However, in the more general case with $\text{output}(i_1, n) \neq \text{input}(i_2, n')$, the code generator has to add a move operation m to the DFG, as shown in Figure 4(b). Operation m moves its input value v_1 along a path in the ISG, from $\text{carrier}(v_1)$ to $\text{carrier}(v_2)$, with v_2 being its output value. Generally, a move operation is implemented by ISG operations in the set $V_I^{\text{move}} \subset V_I$ that contains copy operations to copy values between transitories, read and write operations to access register(file)s, and load and store operations to access memories. The set of ISG operations that are selected by the code generator to implement a particular move operation m, is returned by the function $\text{delivery}(m)$. For the data dependency

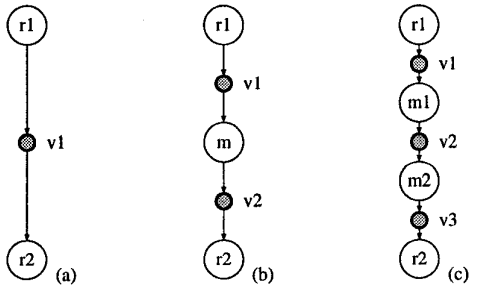

Figure 4: Data dependencies in the DFG : (a) direct data dependency; (b) direct data dependency with move operation; (c) allocated data dependency.

of Figure 4(b), the move operation will be implemented by a series of copy operations, according to following equations :[2]

$$\text{delivery}(m) = \{c_1, c_2, \ldots, c_n\} \subset V_I^{\text{move}} \quad (2)$$

$$\text{carrier}(v_1) = \text{output}(i_1, n) = \text{input}(c_1) = t_0 \quad (3)$$

$$\forall k : 1 \leq k < n : \text{output}(c_k) = \text{input}(c_{k+1}) = t_k \quad (4)$$

$$\text{carrier}(v_2) = \text{input}(i_2, n') = \text{output}(c_n) = t_n \quad (5)$$

$$\{t_0, t_1, t_2, \ldots, t_n\} \subset V_T \quad (6)$$

In Figure 4(c), an *allocated* data dependency is shown. Such a data dependency is bound to a path in the ISG that includes a static storage element. It also holds for an allocated data dependency that $\text{output}(i_1, n) \neq \text{input}(i_2, n')$, but now two move operations m_1 and m_2 are added to the DFG by the code generator. Operation m_1 moves v_1 from $\text{carrier}(v_1) \in V_T$ to $\text{carrier}(v_2) \in V_R \cup V_M$ and m_2 moves v_2 to $\text{carrier}(v_3) \in V_T$. Because v_2 is bound to a static storage element, $\text{delivery}(m_1)$ will now contain a write or a store operation and analogously $\text{delivery}(m_2)$ will contain a read or a load operation. Equations similar to equations (2) to (6) can be written down for each of these delivery functions.

Correctness constraints of bundles. We define a *bundle* as a set of DFG operations that are bound to the ISG and for which following properties hold. Two operations that have a direct data dependency will belong to the same bundle and two operations having an allocated data dependency must be in different bundles. Because of the direct data dependencies, operations in a bundle will be scheduled to execute in the same cycle. Consequently, operations in a bundle are not allowed to have encoding conflicts nor hardware conflicts with each other. The code generator prevents conflicts by changing a direct data dependency into an allocated data dependency, thereby splitting the bundle.

In the remainder of this paragraph, we will introduce definitions to eventually define formal correctness constraints for bundles. Let the resources that an operation $r \in V_O$ uses, be given by $\text{resources}(r) = \{t \in V_T | \exists n : t = \text{output}(\text{type}(r), n)\}$, and by $\text{resources}(m) = \{t \in V_T | \exists c \in \text{delivery}(m) : t = \text{output}(c)\}$ for move operation m. Further, $\text{enabling}(r)$ returns the enabling condition for operation r and $\text{enabling}(m) = \bigcap_{c \in \text{delivery}(m)} \text{enabling}(c)$ the enabling condition for move operation m. Below, we will use the symbol o, meaning either a refined operation r or a move operation m. Finally, we define the predicate $\text{direct}(o_i, o_j)$ to indicate whether $o_i, o_j \in V_O$ have a direct data dependency and $\text{allocated}(o_i, o_j)$ to indicate an allocated data dependency.

[2]For operations that have one input and one output, input() and output() need only one argument.

At this point, we define the correctness constraints for bundles as follows :

$$\forall o_i, o_j \in V_O, o_i \in B_i, o_j \in B_j : \\ \mathsf{direct}(o_i, o_j) \Rightarrow B_i = B_j \tag{7}$$

$$\forall o_i, o_j \in V_O, o_i \in B_i, o_j \in B_j : \\ \mathsf{allocated}(o_i, o_j) \Rightarrow B_i \neq B_j \tag{8}$$

$$\forall o_i, o_j \in B : o_i \neq o_j \\ \Rightarrow \mathsf{resources}(o_i) \cap \mathsf{resources}(o_j) = \emptyset \tag{9}$$

$$\mathsf{enabling}(B) = \bigcap_{o \in B} \mathsf{enabling}(o) \neq \emptyset \tag{10}$$

The last constraint defines the enabling condition of a bundle; its resources are given by $\mathsf{resources}(B) = \bigcup_{o \in B} \mathsf{resources}(o)$.

Conflict constraints between bundles. Because all operations in a bundle are executed in the same cycle, the scheduler can use a conflict model between bundles. The elements of a set of bundles $\mathcal{B}$ are not conflicting with each other, if following two constraints hold :

$$\forall B_i, B_j \in \mathcal{B} : B_i \neq B_j \\ \Rightarrow \mathsf{resources}(B_i) \cap \mathsf{resources}(B_j) = \emptyset \tag{11}$$

$$\bigcap_{B \in \mathcal{B}} \mathsf{enabling}(B) \neq \emptyset \tag{12}$$

Code selection and delayed binding. As said above, in our code generator we bind G_{DFG} onto G_{ISG} in subsequent phases. In a first phase, we decide on which data dependencies will become *direct* data dependencies, or equivalently, we partition the DFG operations into groups, each becoming a bundle eventually.

This phase matches graph patterns in the DFG onto graph patterns in the ISG. The DFG operations are refined according to relation (1) and direct data dependencies are bound, as shown in Figure 4(a) and (b), and according to constraints (2) to (6). The code selection tool may add some move operations to the DFG for this. The allocated data dependencies are not yet bound, but it is verified that each bundle input and output can access static storage. We make certain that the correctness constraints for bundles, constraints (7) to (10), are not violated.

However, several binding alternatives may still be possible. In this phase we only decide on the partitioning of the DFG operations into bundles and we delay the exact binding decision to a subsequent task. For this purpose, we take the design, both the DFG and the ISG, to a higher level of abstraction. For each subgraph in the DFG that is formed by the partitioning above, a new abstract operation $g \in \mathcal{L}$ is created, and the subgraph is replaced by an instance of operation g. For each valid binding of this subgraph, a new operation $b \in V_I \subset \mathcal{L}$ is created. Operation b is inserted in the ISG and is a subtype of operation g in the type hierarchy of $\mathcal{L}$. The enabling condition and the resources of operation b are derived from the original ISG and annotated with b.

In this way we have the same type/subtype relations as depicted in Figure 3, but between larger grain operations, with a DFG and an ISG of much lower complexity. Specific binding possibilities, to be decided on in a subsequent phase, are directly accessible in the library.

Register allocation. In a second phase, the (allocated) data dependencies between bundles are bound. This means that for each operation g an implementation b is chosen and move operations are added at their inputs and outputs to access static storage, while satisfying constraints (7) to (10). In addition to the simple paths considered in Figure 4(c), also paths visiting more than one static storage can be needed, for example to spill a register to memory and to reload it. In the latter case, new bundles are inserted that only consist of single move operations.

We can now summarise two characteristics of the ISG model, which pattern-based models are lacking :

- Its graph structure allows the binding problem of values to be formulated as a *path-search* problem.
- It supports an *incremental* construction of the eventual instructions, by combining elementary actions (either operations or moves) into bundles. By associating resources and enabling attributes with every elementary action, and by defining how these attributes can be combined, an accurate conflict behaviour is available for every partial bundle.

These two characteristics are essential for the register allocation tool, to evaluate the impact of the different routing alternatives for a single data dependence on both the register and timing cost.

Scheduling. After register allocation, all operations and values are bound to the ISG, but they must still be bound to time. During this scheduling phase, for each operation also an instruction is chosen from its enabling condition. The objective is to minimise cycle count, thus as many operations as possible should be scheduled in parallel, while satisfying the conflict constraints (11) and (12).

5 Implementation and results

The retargetable code generator CHESS is implemented according to the concepts developed in Sections 3 and 4. An overview of this retargetable code generation environment can be found in [6]. From the ISG, our simulator

generator CHECKERS automatically derives C++ code implementing an instruction level simulator, by ordering the ISG operations in topological order, guarding them with their enabling conditions and including a bit true behavioural model for each operation in the library $\mathcal{L}$. The resulting simulator is bit and cycle true.

We do not specify the ISG model directly. Instead, it is automatically derived from the high-level processor description language nML [4]. nML consists of an attributed grammar that represents the instruction set hierarchically. The operations to be performed for an instruction and the corresponding encoding are held in the attributes of nML's production rules.

We have modelled several processors in the ISG, including an ASIP tuned to the GSM baseband functions, an ASIP for wave digital filtering and the Analog Devices ADSP-2111 [2], without needing more information than that typically found in a programmer's manual. This is indeed the abstraction level at which a processor is modelled in the ISG. More (implementation) details are unneeded and often not available, either because of confidentiality issues or because the processor simply does not yet exist. Indeed, in hardware/software co-design, retargetable tools are used to provide quality measures on how well hardware and software fit together, before the actual hardware implementation [9].

Currently, we are benchmarking CHESS on these processors. In several cases CHESS yields code that is comparable in quality to hand-crafted code.

6 Conclusions

We presented the ISG processor model, which is the basis of our retargetable code generator CHESS and instruction set simulator CHECKERS. The ISG replaces the pattern base and additional processor views needed in other compilers. Instead, it explicitly models the processor connectivity and it captures the processor parallelism so that both encoding and hardware conflicts are easily checked for. This makes the ISG especially suited as a model to perform register allocation for processors with heterogeneous register structures.

We also explained how we use the ISG to generate code. Subsequent phases in the code generator incrementally define patterns of conflict-free operations that eventually form complete instructions. Thereby, some binding decisions are delayed from one phase to another.

Architectural peculiarities of DSP processors, such as for example residually controlled operations, conditionally executable operations and other special control flow instructions, directly fit in the ISG model as it is presented in this paper. However, some others such as overlapping storage elements (e.g. MR2, MR1, MR0 and MR in the ADSP-2111, often used to convert the data type of an accumulation result to the standard fractional type) and operations with complicated timing profiles, require some extensions.

Currently we are implementing an *aliasing* mechanism that will model overlapping storage elements and will allow to associate different data types with one physical storage element. In the near future we will extend the conflict models in the tools to allow multi-cycle operations and transitories with non-zero delay.

Acknowledgement The authors gratefully acknowledge contributions from Stefan De Troch, who developed the instruction set simulator CHECKERS.

References

[1] A. V. Aho and S. C. Johnson. Optimal code generation for expression trees. *Journal of the Association for Computing Machinery*, 23(3):488–501, July 1976.

[2] Analog Devices. *ADSP-2111, User's Manual*, 1990.

[3] G. Araujo and S. Malik. Optimal code generation for embedded memory non-homogeneous register architectures. In *Proc. of 8th. Int. Symp. on System Synthesis (ISSS)*, Cannes (France), Sept. 1995.

[4] A. Fauth, J. Van Praet, and M. Freericks. Describing instructions set processors using nML. In *Proc. European Design and Test Conf.*, pages 503–507, Paris (France), Mar. 1995.

[5] D. Landskov, S. Davidson, B. Shriver, and P. Mallett. Local microcode compaction techniques. *ACM Computing Surveys*, 12(3):261–294, Sept. 1980.

[6] D. Lanneer, J. Van Praet, A. Kifli, K. Schoofs, W. Geurts, F. Thoen, and G. Goossens. Chess : retargetable code generation for embedded DSP processors. In *Code Generation for Embedded Processors*, pages 85–102. Kluwer academic publishers, Dordrecht, The Netherlands, 1995.

[7] C. Liem, T. May, and P. Paulin. Register assignment through resource classifiction for ASIP microcode generation. In *Proc. IEEE Intl. Conf. on Computer Aided Design*, pages 397–402, Nov. 1994.

[8] L. Nowak and P. Marwedel. Verification of hardware descriptions by retargetable code generation. In *26th ACM/IEEE Design Automation Conference*, 1989.

[9] J. Van Praet, G. Goossens, D. Lanneer, and H. De Man. Instruction set definition and instruction selection for ASIPs. In *Proc. 7th ACM/IEEE Int. Symp. on High Level Synthesis*, pages 11–16, 1994.

[10] V. Živojnović, J. M. Velarde, C. Schläger, and H. Meyr. DSPstone : A DSP-oriented benchmarking methodology. In *Proceedings of ICSPAT*, Dallas,TX, 1994.

[11] B. Wess. Optimizing signal flow graph compilers for digital signal processors. In *Proc. Int. Conf. Signal Proc. Applic. and Technology*, pages 665–670, Dallas, TX, Oct. 1994.