A BDD-based Frontend for Retargetable Compilers

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

In this paper we present a unified frontend for retargetable compilers that performs analysis of the target processor model. Our approach bridges the gap between structural and behavioral processor models for retargetable compilation. This is achieved by means of *instruction set extraction*. The extraction technique is based on a BDD data structure which significantly improves control signal analysis in the target processor compared to previous approaches.[1]

1 Introduction

Commercially available compilers for DSP processors still do not provide sufficient code quality in case of hard real-time constraints. This is mainly due to the fact that modern DSP instruction sets offer a high degree of potential parallelism, but on the other hand incorporate weird restrictions e.g. in register usage. These features cannot be handled by traditional compiler technology. Therefore, machine code generation for DSPs is nowadays mostly done on the assembly level. Because of the obvious drawbacks of assembly-level software development, *retargetable compilation* has gained a lot of interest both in academia and industry. Retargetable compilers read both a HLL algorithm and a description of the target processor and map the algorithm into a machine program for the given target processor.

Several implementations of retargetable compilers are described in the literature. In this paper we focus on the *processor description style* which is accepted by those compilers. Two different approaches have been taken:

Behavioral models: The target processor is described by its instruction set from a *programmer's point of view*, using a specialized language. All instruction set details must be provided by the user. Examples are the CBC compiler [1] that accepts nML descriptions, and the CodeSyn compiler [2].

Structural models: The target processor is described by an RT-level netlist in a HDL. The complete datapath and controller structure are part of the processor model. Examples are the compilers MSSV [3] and MSSQ [4].

The decision which kind of modelling style is the best depends on the target processor. When the target is a standard DSP, the instruction set is fixed, and a behavioral model should be used. In case the target processor is an ASIP with a non-fixed instruction set, a structural model is more adequate. Other applications might even require a mixed model comprising only a few hardware modules with complex behavior, e.g. a processor could be described as a netlist consisting of one controller module and one datapath module.

Previous retargetable compilers accept only one certain style. In order to overcome this restriction we propose a unified frontend for retargetable compilers which accepts processor models in either style (behavioral, mixed, or structural) and which extracts the instruction set from the processor model. The extracted instruction set is independent from the description style and forms the basis for the actual code generation phase of compilation. In case of pure behavioral models the extractor essentially transforms the given instruction set into an internal format. In case of mixed or pure structural models the instruction set is derived from the structure. The extraction tool is part of a retargetable compiler system for DSPs and ASIPs, which is currently under development.

Since extraction in case of pure behavioral models is trivial we assume throughout this paper that the target processor is given by a structural or mixed model. Compared to the approach presented in [5], several deficiencies have been eliminated: The new extractor accepts processor descriptions in the powerful MIMOLA 4.1 HDL [6], and condition analysis is based on a uniform BDD data structure.

The rest of the paper is organized as follows: Section 2 briefly describes the construction of a graph model from the processor HDL model, which forms the basis for the extraction process. Extraction of microoperations proceeds in two phases explained in sections 3 and 4. The paper end with first experimental results and conclusions.

2 Graph representation of the target processor

The MIMOLA model of the target processor consists of a set of *modules* and their *interconnections*. Modules may either be *structural* (i.e. consist of interconnected submodules) or *behavioral* (i.e. hide their internal structure). We assume that the "outermost" module describing the complete target processor is a structural one. Its submodules in turn may be either structural or behavioral. The textual MIMOLA model of the target processor is transformed into a flat graph representation as depicted in fig. 1. The graph nodes represent the set of behavioral modules. Each module M_i has a set of *ports* $\{p_{i1}, \ldots, p_{im_i}\}$. Each port has an associated mode (IN, OUT, INOUT) implying the direction of dataflow at this port. Edges represent port connections. The graph

[1] This work has been partially supported by ESPRIT BRA project 9138 (CHIPS)

representation is used to analyze dataflow between behavioral modules.

3 Analysis of local module behavior

Behavioral modules in MIMOLA are described in a procedural manner, based on a comprehensive set of primitive operators. The module interface is described as a port list and the module behavior within a *concurrent block*, e.g.:

```
MODULE m1 (IN i1, i2: (15:0); IN ctr: Bit;
  OUT o1, o2: (15:0));
VAR v: (15:0);
CONBEGIN
   IF i1.(7:0) > i2.(15:8)
    THEN o1 <- 1 ELSE o1 <- 0;
   IF ctr THEN v := i1;
   o2 <- i1 + i2;
CONEND;
```

The CONBEGIN/CONEND block of a behavioral module contains a set of concurrent statements describing assignments either to module output ports or to internal variables. IF- and CASE-constructs can be used to model (arbitrarily nested) conditional statements.

The local behavior of a module M_i can be represented by a set of concurrent *assignments* A_i. An assignment $a_{ij} \in A_i$ is a triple $a_{ij} = (d_{ij}, e_{ij}, c_{ij})$ where d_{ij} is the *destination* (a storage cell or an output port), e_{ij} is an *expression* (a signal which is assigned to d_{ij}), and c_{ij} is a *condition* (which must be fulfilled to execute the assignment). The condition c_{ij} can be considered as a Boolean function. In case of an unconditional assignment (`o2 <- i1 + i2` in the example), c_{ij} is the constant 1 function.

Whereas it is easy to represent destinations and expressions, it is crucial for a retargetable compiler working on netlist descriptions to include a careful analysis of assignment conditions. Analysis of assignment conditions have also been discussed in [4, 5]. However, these methods imply restrictions concerning either the possible sources of conditions or compatibility checking between those. The representation of conditions proposed in this paper is based on BDDs. This choice has been made for the following reasons:

1) BDDs provide a uniform data structure that supports many important operations during instruction set extraction. We use the well-known BDD package [10].

2) During code generation experiments with MSSQ it turned out that conveniently modelling complex processors often requires a *bit-level* condition representation, e.g. concatenated control inputs for modules should be possible. Single-bit conditions are easily represented by BDD variables.

3) A BDD representation of conditions permits *minimizing the effects of syntactic variances* in processor descriptions. The importance of this feature has already been recognized in the context of High-Level Synthesis [11].

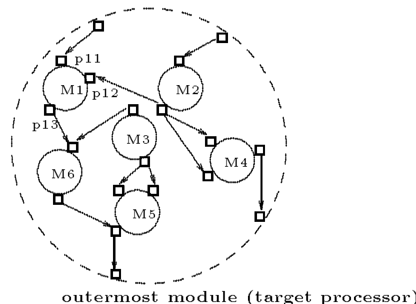

Figure 1: *Graph representation of the target processor*

Representations for nested assignment conditions have also been proposed in the area of High-Level Synthesis [8, 9]. For instruction set extraction, however, Boolean manipulation of conditions is required, which is better supported by BDDs.

Construction of assignments in case of unconditional statements like `v := i1 + i2;` is trivial. For this statement the assignment $a = (v, i1 + i2, TRUE)$ is constructed, where $TRUE$ represents the constant 1 function. Destinations and expressions are simply represented by their identifiers resp. arithmetical or logical expressions on these. The main problem is to correctly extract and represent all conditions in statements. The following two subsections describe the extraction of assignments in case of conditional statements. The description is based on MIMOLA language elements but the concepts are language independent.

3.1 IF-conditions

IF-conditions occur in MIMOLA statements of the form

```
IF <cond_expr> THEN <then_statement>
      [ELSE <else_statement>]
```

The conditional expression `<cond_expr>` may either refer to a module input port or a variable (e.g. `IF enable THEN` ...) or may denote a complex expression (e.g. `IF (i1 > i2) AND (i3 <= i4) THEN` ...).

From the *compiler's point of view*, three kinds of assignment conditions exist:

1. **I-conditions** that refer to necessary values of certain instruction bits.
2. **M-conditions** that refer to a certain machine state, i.e. the value of registers or memory cells. M-conditions occur in case of residual control or status registers.
3. **D-conditions** that are dynamically evaluated at runtime, e.g. a signal comparison.

The basic idea in our BDD-based approach is to represent conditions by Boolean functions and to classify their Boolean variables with respect to the three types I, M, and D. This classification provides the necessary information for the compiler's code generation phase:

1) A Boolean variable I_k represents bit no. k in the

"current" instruction word.
2) A Boolean variable M_{rk} represents bit no. k in register r.
3) A Boolean variable D_e represents a conditional expression e that is evaluated at runtime.

If `<cond_expr>` refers to a module input port, its classification can only be done after looking beyond the module boundaries. Since in the first phase we consider modules only *locally*, we introduce a fourth class of Boolean variables (**P-variables**) which temporarily represent bits of module input ports. P-variables are replaced in the second phase. The following algorithm constructs a set of assignments from an IF-statement.

Algorithm *extract_assignments_from_IF_statement*

1) Recursively transform `<then_statement>` into a set of assignments $A_T = \{a_1, \ldots, a_n\}$ The result in general is a set of assignments, since `<then_statement>` might be a set of (possibly nested conditional) concurrent statements.

2) If the IF-statement has an ELSE-part, recursively transform `<else_statement>` into a set of assignments $A_E = \{b_1, \ldots, b_m\}$
3) Transform `<cond_expr>` into a Boolean function F (represented by a BDD):

- If `<cond_expr>` refers to bit number k of a module variable v: `<cond_expr>` is an M-condition. Create a new Boolean variable M_{vk} and set $F := M_{vk}$.
- If `<cond_expr>` refers to bit number k of a module input port p: The classification of `<cond_expr>` must be postponed. Create a new Boolean variable P_{pk} and set $F := P_{pk}$.
- If `<cond_expr>` denotes a complex expression $e = op(e_1, e_2)$, where op is an operator and e_1, e_2 are expressions:
  - If op is a Boolean operator (AND, NAND, OR, NOR, XOR, XNOR), then decompose e: Recursively construct the Boolean functions F_{e_1}, F_{e_2} for e_1, e_2 and set $F := op(F_{e1}, F_{e2})$.
  - If op is a non-Boolean operator, leave the expression e untouched as a D-condition. Create a new Boolean variable D_e and set $F := D_e$.

4) For each assignment $a_i = (d_i, e_i, c_i) \in A_T$ construct the assignment $a'_i = (d_i, e_i, c_i \wedge F)$
5) For each assignment $b_j = (d_j, e_j, c_j) \in A_E$ construct the assignment $b'_j = (d_j, e_j, c_j \wedge \overline{F})$
6) Return the assignment set $\bigcup_i a'_i \quad \cup \quad \bigcup_j b'_j$

3.2 CASE-conditions

Multiple module control signals can be bundled for sake of convenience, and the behavior can be expressed using CASE-statements. A MIMOLA CASE-statement has the format:

```
CASE <sel> OF
  N_11,...,N_1k_1:  <statement_1>
  ...
  N_m1,...,N_mk_m:  <statement_m>
  ELSE: <else_statement>;
END;
```

The selector `<sel>` refers to a module input port or variable of s bits width, and the numbers N_{ij} represent possible values of `<sel>`. If `<sel>` is equal to one $N_{ij}, j \in \{1, \ldots, k_i\}$ then `<statement_i>` is executed. Otherwise, `<else_statement>` is executed. The following algorithm constructs an assignment set from a CASE-statement.

Algorithm *extract_assignments_from_CASE_statement*

1) Let the bits of `<sel>` be represented by Boolean variables $v_1, \ldots, v_s$. If `<sel>` refers to a module input port, then $v_1, \ldots, v_s$ are of type P, otherwise (if `<sel>` refers to a module variable) they are of type M.
2) Assuming the values N_{ij} are given as bitstrings $N_{ij} = (n^1_{ij}, \ldots, n^s_{ij})$ compute the Boolean functions F_{ij} so that

$$\begin{array}{rrcl} & F_{ij}(v_1, \ldots, v_s) & = & 1 \\ \Leftrightarrow & (v_1, \ldots, v_s) & = & (n^1_{ij}, \ldots, n^s_{ij}) \end{array}$$

Thus, F_{ij} represents the *minterm* for N_{ij}.
3) For each `<statement_i>`, $i \in \{1, \ldots, m\}$ recursively compute the assignment set A_i.
4) For each assignment set A_i construct the assignment set A'_i by replacing the conditions c in the assignments in A_i by $c \quad \wedge \quad \bigvee_j F_{ij}$
5) For the ELSE-part of the CASE statement, compute the Boolean function $E = \bigwedge_{i,j} \overline{F_{ij}}$ which covers all cases not covered by the N_{ij}'s. Recursively construct the assignment set A_E for `<else_statement>` and the assignment set A'_E by replacing the conditions c in the assignments in A_E by $c \quad \wedge \quad E$
6) Return the assignment set $\bigcup_i \quad A'_i \quad \cup \quad A'_E$

With the above algorithms, an arbitrary, nested statement of any type (unconditional, IF, CASE) can be transformed into a set of assignments. This transformation is applied to each statement in the module behavioral description. The assignment conditions are Boolean functions on variables of type M, D, or P.

The uniform BDD representation of conditions reduces the *effects of syntactic variances*. For instance, the following equivalent formulations of a condition depending on two bits[2] result in the same condition ("$x1 \wedge x2$"):

```
IF x1 THEN IF x2 THEN ...
IF x2 THEN IF x1 THEN ...
IF x1 AND x2 THEN ...
CASE x1!!x2 OF %11: ...
```

The BDD representation of conditions also enables checking of concurrent assignments to the same destination for mutual exclusion.

[2] !! denotes concatenation

4 Composition of μ-operations

After the first phase, each module M_i is represented by its local set of concurrent assignments. These assignments are now used to compose complete *microoperations*. A microoperation m is an assignment (d, e, c), where

1) the destination d is a register, a storage cell, or an external output
2) the expression e only has registers, storage cells, external inputs, or hardwired constants as arguments
3) the Boolean variables in the condition c only refer to instruction bits, register bits, or expressions, i.e. they are of type I, M, or D.

Composing microoperations requires expanding local assignments across module boundaries, so that all *references to internal module ports are resolved.* Expansion stops at sequential modules, and at external inputs and hardwired constants. Two expansion procedures for expressions resp. conditions are used. Expansion works on the graph model of the target processor (see section 2). The following subsections describe the expansion procedures.

4.1 Expansion of expressions

Expression expansion combines expressions to more complex ones by analysis of the processor interconnect structure. For instance, a multiply-accumulate chain would be detected in this phase. An expression e is either

1) a *complex expression* of the form $e = op(e_1, \ldots, e_n)$, where $e_1, \ldots, e_n$ are expressions and op is an operator, or
2) an *indexed expression* (denoting a storage cell) of the form $e = e_1[e_2]$, where e_1, e_2 are expressions, or
3) a *simple expression* which denotes a module input port, a register, a hardwired constant, or an bit index subrange of these.

The basic step in expression expansion is to *replace* module input ports in expressions by those expressions which can be switched to these ports via possible data routes within the structure. Since in general a number of possible routes exist, expression expansion introduces new conditions, which select one of these possibilities. The possible expressions which can be switched to a certain module port together with the corresponding conditions are determined by the graph model and the assignment sets found during the first extraction phase. We explain expression expansion using a simple example (fig. 2).

Consider an expression $NOT(p)$, where p denotes an input port of a certain module M. Using the graph model, one can find p's predecessor port q in another module M', which is connected to p. Let A_q be the set of assignments (q, e_i, c_i) in M' with destination q. Then, p can be replaced by each expression e_i, presuming that condition c_i holds. Therefore, expanding an expression in general delivers a set of possible expressions together with the corresponding conditions. In the example, expansion of $NOT(p)$ delivers the expression/condition set

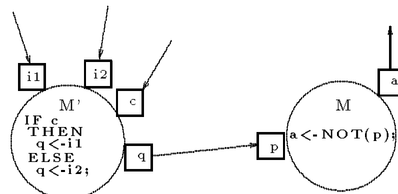

Figure 2: *Expansion of expressions using the graph model*

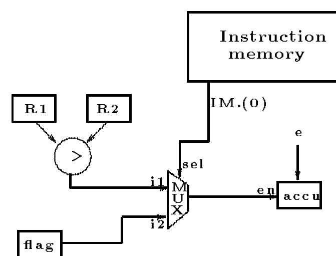

Figure 3: *Expansion of conditions*

$$\{(NOT(i1), P_{c,0}), \quad (NOT(i2), \overline{P_{c,0}})\}$$

The expressions e_i are expanded recursively. Recursion stops at registers, memories, external inputs, and hardwired constants.

The extractor also handles tristate busses, which require a special treatment. When a tristate bus is reached during expansion, the extractor tries to set all unused bus drivers to a "Z" mode, which imposes additional conditions. Expansion fails if a bus driver cannot be set to "Z" mode by any condition.

4.2 Expansion of conditions

The conditions of assignments extracted in phase 1 depend on Boolean variables of type M, D, and P. P-variables represent module input port bits. The purpose of condition expansion is to replace the P-variables by Boolean functions so that expanded conditions only depend on variables of the types M, D, and I. This replacement is equivalent to the composition of Boolean functions. Replacement of P-variables is done by recursively traversing the graph model of the target processor and updating conditions on-the-fly. Recursion stops if the instruction memory, a register, a dynamic condition, or a hardwired constant is reached. We explain the basic idea using the simple example in fig. 3. Since all relevant signals in the example are one bit wide, we omit the second variable index denoting the bit index for sake of simplicity.

Destination register `accu` can be loaded with some expression `e`. The assignment condition is that the enable input bit `en` is set to 1. Thus, the Boolean function F representing the condition is initially

$$F = P_{en}$$

`en` is traced back to the output of multiplexer `MUX`, which

passes its input signal `i1` if `sel` is 0, or `i2` if `sel` is 1. The bit signal `sel` is found to be equal to bit 0 of the instruction memory, so that

$$F = (P_{i1} \wedge \overline{I_0}) \vee (P_{i2} \wedge I_0)$$

Tracing back `i1` and `i2` yields the dynamic condition `R1 > R2` and bit no. 0 of register `flag`, respectively. Thus, F is expanded to

$$F = (D_{R1>R2} \wedge \overline{I_0}) \vee (M_{flag} \wedge I_0)$$

which only depends on M-, D-, and I-variables. F reveals two versions for the assignment `accu := e`:

1) `R1` is greater than `R2` and instruction bit no. 0 is 0.

2) Register `flag` contains the value 1 and instruction bit no. 0 is 1.

These versions are the basis for the instruction selection and scheduling phase of code generation. Versions correspond to *implicants* of the Boolean function F. Normally, an arbitrary complete set of implicants is is directly derived from the BDD representation of F. Optionally, the user can enforce computation of *prime implicants*. This feature guarantees versions without unnecessary restrictions which might obstruct code compaction.

4.3 Expansion of assignments

Using the above procedures, all possible microoperations for the given processor can be extracted. For each clocked assignment $a = (d, e, c)$ the following steps are performed:

1) Expand the expression e resulting in a set of expression/condition pairs
$EC = \{(e_i, c_i) | i \in \{1, \ldots, k\}\}$.
2) Expand the condition c resulting in a condition c'.
3) Expand the conditions c_i resulting in conditions c'_i.
4) For each $(e_i, c_i) \in EC$ construct the microoperation $m_i = (d, e_i, c'_i \wedge c')$

Expansion of assignments may yield constant *FALSE* conditions $c'_i \wedge c'$, i.e. the corresponding microoperation is invalid due to some encoding restrictions and can be excluded from the instruction set. Thus, encoding restrictions are automatically derived from the structure. The result of assignment expansion is the set of valid microoperations for the specified processor.

5 Experimental results

Table 1 shows experimental results obtained so far with the new instruction set extractor. The CPU seconds (on a SPARC-20), including setup times, are listed together with the number of extracted microoperations. In order to outline the complexity of the circuits, the number of HDL text lines and RT-level modules are given. The target processors include an industrial ASIP (bassboost), a processor (mano) described in [12], and an off-the-shelf DSP processor [7]. In the latter case a mixed structural/behavioral processor model has been used. The experiments indicate that our new approach to instruction set extraction is fast enough to be integrated within a retargetable compiler system. Compared to [5] the runtime could even be significantly reduced.

Processor	CPU	μ-ops	HDL lines	modules
asip	1	38	221	20
bassboost	3	26	416	29
p1	7	131	278	14
p2	1	44	291	10
mano	2	124	518	19
ref	4	82	207	13
TMS320C25	64	572	2480	4

Table 1: *Experimental results*

6 Conclusions

The main contributions of this paper are twofold. Firstly, a unified frontend for retargetable compilers was presented that accepts target processor models in either style (behavioral, structural, mixed) and extracts the instruction set from the processor model. The resulting microoperation set abstracts from the processor model and provides a suitable basis for the later phases of retargetable compilation.

Secondly, we discussed the applicability of BDDs in condition analysis for retargetable compilation. By using a BDD-based representation several subproblems could be strongly simplified compared to previous approaches, e.g. checking for compatibility and mutual exclusion, detecting encoding restrictions and reducing the effects of syntactic variances in processor descriptions.

References

[1] A. Fauth, A. Knoll: Translating signal flowcharts into microcode for custom digital signal processors, Proc. ICSP, 1993

[2] C. Liem, T. May, P. Paulin: Instruction-set matching and selection for DSP and ASIP code generation, Proc. ED & TC, 1994

[3] P. Marwedel: Tree-Based Mapping of Algorithms to Predefined Structures, Proc. ICCAD, 1993, pp. 586-593

[4] L. Nowak, P. Marwedel: Verification of Hardware Descriptions by Retargetable Code Generation, Proc. 26th DAC, 1989, pp.441-447

[5] R. Leupers, P. Marwedel: Instruction set extraction from programmable structures, Proc. EURO-DAC, 1994, pp. 156-161

[6] P. Marwedel, et al.: The MIMOLA language version 4.1, Technical Report, available from the authors

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

[8] K. Wakabayashi, H. Tanaka: Global scheduling independent of control dependencies based on condition vectors, Proc. 29th DAC, 1992, pp. 112-115

[9] M. Rim, R. Jain: Representing conditional branches for high-level synthesis applications, Proc. 29th DAC, 1992, pp. 106-111

[10] K. Brace, R. Rudell, R. Bryant: Efficient implementation of a BDD package, Proc. 27th DAC, 1990

[11] V. Chaiyakul, D. Gajski, L. Ramachandran: High-level transformation for minimizing syntactic variances, Proc. 30th DAC, 1993

[12] M.M. Mano: Computer System Architecture, Prentice Hall International Inc., 3rd Ed., 1993