HDL-based Modeling of Embedded Processor Behavior for Retargetable Compilation

Rainer Leupers
University of Dortmund
Dept. of Computer Science 12
44221 Dortmund, Germany
email: leupers@ls12.cs.uni-dortmund.de

Abstract—The concept of retargetability enables compiler technology to keep pace with the increasing variety of domain-specific embedded processors. In order to achieve user retargetability, powerful processor modeling formalisms are required. Most of the recent modeling formalisms concentrate on horizontal, VLIW-like instruction formats. However, for encoded instruction formats with restricted instruction-level parallelism (ILP), a large number of ILP constraints might need to be specified, resulting in less concise processor models. This paper presents an HDL-based approach to processor modeling for retargetable compilation, in which ILP may be implicitly constrained. As a consequence, the formalism allows for concise models also for encoded instruction formats. The practical applicability of the modeling formalism is demonstrated by means of a case study for a complex DSP.

1 Introduction

As a result of the increasing diversity of embedded processors, retargetable compilers have received renewed interest [1]. Retargetable compilers are cross-compilers that can be quickly adapted to different target instruction sets without modifying the kernel of the compiler source code. Retargeting is performed by providing the compiler with a – usually textual – model of the processor for which code is to be generated. Retargetable compilers are particularly important in the context of application-specific processors (ASIPs), for which normally no target-specific compiler support is available.

Recent activities in retargetable compilation for embedded processors aim at generating compilers from processor models in uniform, user-editable languages that also capture instruction-level parallelism (ILP). In most of this work machine instruction behavior is described as a collection of RT templates, which represent atomic processor operations. During code generation, instances of different RT templates may be packed into the same instruction word, i.e., some RTs can be executed in parallel. Potential ILP is constrained either by the resource usage of RTs or by the instruction encoding scheme.

In such a processor modeling style, ILP is implicit, because there is an a priori assumption of unrestricted parallelism between RT templates. Constraints on ILP are made explicit through enumeration of invalid template combinations. This is mainly useful for instruction formats with a small number of constraints, i.e., for horizontal VLIW-like formats, where there is a dedicated instruction field for each of the parallel functional units in the data path. On the other hand, encoded instruction formats with very restricted ILP may require the enumeration of a large amount of constraints.

The purpose of this paper is to present a hardware description language (HDL) based processor modeling formalism for retargetable compilation. The key feature of the proposed formalism is a flexible notion of instruction behavior and parallelism. By means of appropriate modeling language constructs, ILP may be modeled explicitly, while all ILP constraints are implicit and can be automatically extracted from the model. However, also the abovementioned converse modeling style with implicit ILP and explicit constraints is supported. As a result, concise models can be developed both for horizontal and encoded instruction formats.

The organization of this paper is as follows. In section 2, we discuss several recent processor modeling formalisms. In section 3, we outline our approach to processor modeling for retargetable compilation. Since the details are best described by an example, we present a case study for a real-life DSP in section 4. Concluding remarks are given in section 5.

2 Related work

Several compilers for embedded processors have been designed recently, which make use of tool-specific processor description formalisms, mainly intended for internal use in the compiler [2, 3, 4, 5, 6]. The compiler generation approach used in [7] does support user retargetability, but it is based on a collection of translation rules instead of a uniform modeling language.

The MSSQ compiler [8] operates on RT-level processor models in the MIMOLA HDL, which resemble structural VHDL. However, in certain cases this may require knowledge of hardware details which are not available to the compiler user. MSSQ also handles horizontal and encoded instruction formats, but specification of the latter demands for insertion of (possibly large) instruction decoders into the netlist in order to inhibit generation of invalid parallel instructions.

A completely different modeling paradigm underlies the nML language developed at TU Berlin, which is the
The HDL-based formalism

3.1 Component behavior

Unlike in most other approaches, we do not use a notion of an instruction set or of RT templates in processor models, but models are supposed to be composed of a set of one or more components working in parallel. Each component $C = (I, R, B)$ consists of an interface $I$ (specified as an I/O port list), a set of local registers or register files $R$, and a behavior $B$. In case of multiple components a list of connections and busses between component I/O ports are specified. The behavior $B = \{A_1, \ldots, A_n\}$ is a set of one or more concurrent conditional assignments $A_i$. The syntax for component behavior corresponds to the MIMOLA language used by the MSSQ compiler [8]. The modeling concept, however, is independent of a certain language, and using VHDL, which offers the same language constructs, would obviously be possible as well. The main enhancement, as compared to MSSQ, is that not only RTL structural models are permitted, but also purely behavioral, instruction-level models as frequently found in processor reference manuals. The syntax of component behavior specification is as follows:

```
<behavior> ::= <conditional behavior> | <assignment>
<conditional behavior> ::= CASE <value> OF <tagged behavior>+ [ ELSE <behavior> ] END |
_IF <value> THEN <behavior> [ ELSE <behavior> ]_
<tagged behavior> ::= <constant> : <behavior>
<assignment> ::= <register destination> := <value> | <port destination> := <value>
)value> ::= <conditional value> | <expression value>
<conditional value> ::= CASE <value> OF <tagged value>+ [ ELSE <value> ] END |
_IF <value> THEN <value> ELSE <value>
<tagged value> ::= <constant> : <value>
<port destination> ::= an output port of the component interface |
<register destination> ::= a member of the local register set R |
<expression value> ::= an input port of the component interface |
```

The semantics of a behavior is, that an assignment is enabled, if and only if all execution conditions attached by IF or CASE constructs evaluate to "true" in the specific instruction cycle. An instance of each enabled assignment is executed once per machine instruction cycle.

All assignments enclosed in a CONBEGIN ... CONEND construct are concurrent. Assignments to sequential components ("<register destination> := ...", which are supposed to be synchronized by a global clock, are executed with a unit delay. Assignments to ports ("<port destination> := ...") have zero delay. An unconditional assignment corresponds to an RT template in other processor modeling languages. Available unary and binary operators for expressions include all common arithmetic, logical, and shift operators, as well as special operators for sign or zero extension and concatenation. The "subrange" construct enables selection of a certain bit index subrange from an argument value.

3.2 Analysis of ILP constraints

The analysis of ILP constraints in the HDL processor model is based on manipulation of Boolean functions that represent the execution conditions for assignments. Each bit line contributing to controlling a component behavior is represented as a Boolean variable. For each possible assignment, a Boolean function is constructed that evaluates to true exactly for those variable bindings that enforce that assignment. Two assignments can be executed in parallel, if their execution conditions can be simultaneously satisfied, i.e., their logical conjunction is not a zero constant.

Any assignment may have multiple occurrences in a behavior, each with different execution conditions. In this case, the logical disjunction of the different conditions is computed, before ILP constraints w.r.t. other assignments are tested. Different execution conditions can be considered as alternative versions of machine operations. The notion of alternative versions is important, since on some DSPs ILP constraints cannot be simply represented as a binary relation between operations.

For efficiency reasons, the analysis of ILP constraints via Boolean function manipulation is based on binary decision diagrams (BDDs). A detailed description is given in [9].

4 Case study: TI TMS320C25

This section describes details of a behavioral model for a complex standard DSP, the Texas Instruments TMS320C25, which shows a strongly encoded instruction format. Due to its very restricted ILP, we model the 'C25 in a purely behavioral style as a single component.

2On the TI TMS320C25 DSP, for instance, an accumulate operation can only be executed in parallel to an address register increment, if a third operation [load T register or multiply] is executed in parallel.
4.1 Instruction format

The 'C25 has 16-bit instructions with an extremely flexible format. For some instructions, the 16 bits are fully occupied by the opcode. Other instructions have opcode widths between 3 and 14 bits, while using the remaining bits as immediate operands. In total, 12 named bit fields may occur in the different instructions, which leads to the following instruction format declaration:

```plaintext
TYPE Instruction = */16-bit instruction format /*

FIELDS
AGU: (6:4); /* AGU control field */
AR: (10:8); /* address register index */
NAR: (3); /* address register notify flag */
INDIRECT: (7); /* direct/indirect addressing flag */
DMA: (6:0); /* data memory address */
SHIFT4: (11:8); /* 4-bit shift value */
SHIFT3: (10:8); /* 3-bit shift value */
K3: (7:0); /* 8-bit immediate operand */
PM: (1:0); /* product mode */
opcode: (15:0); /* instruction opcode */

END;
```

The "FIELDS" construct (which resembles C structures or unions) in the instruction format declaration is used to assign names to certain index subranges of a bit vector. The context-dependent use of instruction bits is reflected in the multiple overlap of the fields. The use of some of these fields will be shown later.

4.2 Component specification

The 'C25 is modelled as a single component with all processor registers being local to that component. The local registers include data registers (e.g., the accumulator), control registers (e.g., the program counter), and AGU registers. Furthermore, there are mode registers, whose states account for the current arithmetic operation mode. The overall shape of the model, including local register declarations, is the following:

```plaintext
MODULE TMS250C25 ( ... ); /* interface declaration */

BEHAVIR IR IS
VAR
IR: Instruction; /* instruction register */
PC: Word; /* program counter */
BO: ARRAY[0..255] OF Word; /* co-chip memory */
AR: ARRAY[0..7] OF Word; /* 8 x 16 addr register file */
ARP: (2:0); /* 3-bit addr register pointer */
DF: (8:0); /* 9-bit memory page pointer */
ACCU: LongWord; /* accumulator register */
PR: LongWord; /* product register */
TR: Word; /* multiplier register */
SRM: Bit; /* sign extension node register */
PH: (0:0); /* 2-bit product node register */

CONBEGIN
/* instruction behavior, see next section */
END;
```

4.3 Instruction behavior

The main execution condition for all instructions is the instruction opcode. Therefore, the processor behavior can be modelled as a single "large" CASE statement, where the selector is the opcode field of the instruction register IR:

```plaintext
CASE IR.opcode OF
  <opcode n>: <behavior n>
  ...;
END;
```

For each value of IR.opcode a different behavior is specified. However, as mentioned earlier, different instructions have a different number of significant bits in the opcode. Therefore our modeling syntax permits specification of don't care conditions. As an example, consider the "multiply immediate" instruction MPYK which has the following format:

```plaintext
MPYK
  PR := TR * IR.K13; /* multiply TR by immediate constant */
  PC := IRK PC; /* increment program counter */
END;
```

Next, we consider modeling of instructions with a more complex behavior. The "add to accumulator with shift" instruction ADD may appear in the following two formats:

```plaintext
ADD
  PC := TR + IR.SHI FT4; /* add to accumulator */
END;
```

Bits 11 down to 8 (IR.SHIFT4) specify a shift value for an argument taken from memory. Bit 7 (IR.INDIRECT) selects direct (0) or indirect (1) addressing and decides on the interpretation of bits 6 down to 0. If IR.INDIRECT = 0, then bits 6 down to 0 (IR.DMA) are taken as an offset of a direct memory address, which is appended to the page pointer DP. Otherwise, if IR.INDIRECT = 1, then the following AGU operations are encoded: Bits 6 down to 4 (IR.AGU) specify a parallel address computation in the AGU, and bit 3 (IR.NAR) determines whether the address register pointer ARP is updated. If IR.NAR = 1, then ARP is assigned the 4-bit value in bits 2 down to 0 (IR.Y). Furthermore, the behavior of ADD depends on the sign extension flag SXK. In total, ADD is modelled as follows:

```plaintext
CASE IR.opcode OF
  <opcode n>: <behavior n>
  ...;
END;
```

Since only the first 3 bits are opcode bits, we specify the behavior with don't cares in the 13 least significant bits of the CASE tag constant.

```plaintext
CASE IR.opcode OF
  010xxxxxxxxx: MPYK instruction */
CONBEGIN
PR := TR * IR.K13; /* multiply TR by immediate constant */
PC := IRK PC; /* increment program counter */
CONEND;
```

The "add to accumulator with shift" instruction ADD may appear in the following two formats:

```plaintext
ADD
  PC := TR + IR.SHI FT4; /* add to accumulator */
END;
```
The CONBEGIN/CONEND construct makes parallelism between the data path and the address generation unit explicit. A number of further 'C25 machine instructions use the same encoding of the addressing behavioral descriptions of these, the same AGU-related behavior down to 0 in the instruction word. In the instruction unit explicit. A number of further 'C25 machine instructions use the above model, the C25 instruction LTA, which loads register TR and accumulates a previous product stored in register PR.

4.4 Results

Knowledge of ILP constraints is necessary to prevent invalid parallelization of RTs generated by a compiler. Due to the small instruction word length of the 'C25, the Boolean functions required for ILP constraint analysis depend on few variables: 16 variables for instruction bits, as well as one variable for each mode register bit. For the 'C25 model, extraction of all ILP constraints using BDDs takes 41 CPU seconds on a SPARC-20. Even though horizontal instruction formats require many more Boolean variables due to very long instruction words, most of these variables are don't care for each specific RT operation. Since don’t cares are not contained in the BDD representations of Boolean functions, efficiency is preserved also for horizontal formats.

In order to demonstrate the applicability to code generation, we have used the 'C25 model as an input to Rcomp [13], a retargetable compiler for fixed-point DSPs. Using the above model, Rcomp was able to compile the DSPStone benchmarks [14]. The model size for the 'C25 DSP is approximately 900 lines of MOLDA HDL code. This appears to be rather concise as compared to other approaches. An nML model developed for a 'C25 extension (the TMS320C50) has been reported [15], which is about twice as large. A "C25 RT-level netlist model, that has been developed for use with the MSSQ compiler [8], consists of more than 5,000 lines of code.

5 Conclusions

In this paper we have presented a flexible HDL-based processor modeling formalism for use with retargetable compilers for embedded processors. We have exemplified, that our modeling formalism is capable of concisely describing the complex instruction sets of realistic embedded processors. This includes the capture of ILP, as well as architectural peculiarities like mode registers. The modeling formalism has been successfully applied to retargetable code generation for DSPs. With respect to processor simulation, it is important to note that the modeling formalism only makes use of language constructs which are contained in most hardware description languages. This implies that the use of standard HDL simulators is possible.

References