Code Generation For Embedded Processors

Peter Marwedel
Technische Universität Dortmund

CODE GENERATION FOR EMBEDDED PROCESSORS

CODE GENERATION FOR EMBEDDED PROCESSORS

EDITED BY

Peter MARWEDEL
Universität Dortmund
Dortmund, Germany

■

Gert GOOSSENS
IMEC
Leuven, Belgium

KLUWER ACADEMIC PUBLISHERS
Boston/London/Dordrecht

CONTENTS

CONTRIBUTORS

Ulrich Bieker
Universität Dortmund
Dortmund, Germany

Marco Cornero
DIBE – Universita di Genova
Genova, Italy

Henk Corporaal
Technische Universiteit Delft
Delft, The Netherlands

Srinivas Devadas
MIT
Cambridge, Mass., U.S.A.

Andreas Fauth
Technische Universität Berlin
Berlin, Germany

Mahadevan Ganapathi
Synopsys
Mountain View, Calif., U.S.A.

Werner Geurts
IMEC
Leuven, Belgium

Gert Goossens
IMEC
Leuven, Belgium

Kurt Keutzer
Synopsys
Mountain View, Calif., U.S.A.

Augusli Kifli
IMEC
Leuven, Belgium

Michel Langevin
GMD
Birlinghoven, Germany

Dirk Lanneer
IMEC
Leuven, Belgium

Stan Liao
MIT
Cambridge, Mass., U.S.A.

Clifford Liem
INPG
Grenoble, France

Sharad Malik
Princeton University
Princeton, NJ, U.S.A.

Peter Marwedel
Universität Dortmund
Dortmund, Germany

Farhad Mavaddat
University of Waterloo
Waterloo, Ont., Canada

Heinrich Meyr
RWTH
Aachen, Germany

Alex Nicolau
U.C. Irvine
Irvine, Calif., U.S.A.

Pierre Paulin
SGS-Thomson
Grenoble, France

Sebastian Ritz
RWTH
Aachen, Germany

Wolfgang Schenk
Universität Dortmund
Dortmund, Germany

Koen Schoofs
IMEC
Leuven, Belgium

Filip Thoen
IMEC
Leuven, Belgium

Kees Vissers
Philips Research
Eindhoven, The Netherlands

Paul Van Oostende
Alcatel-Bell
Antwerpen, Belgium

Johan Van Praet
IMEC
Leuven, Belgium

Albert Wang
Synopsys
Mountain View, Calif., U.S.A.

Bernhard Wess
Technische Universität Wien
Wien, Austria

Tom Wilson
University of Guelph
Guelph, Ont., Canada

1

RETARGETABLE COMPILATION OF SELF-TEST PROGRAMS USING CONSTRAINT LOGIC PROGRAMMING

Ulrich Bieker

Department of Computer Science,
University of Dortmund, D-44221 Dortmund, Germany

ABSTRACT

This chapter presents a retargetable code generator specialized in the compilation of self-test programs and exploiting new techniques from Constraint Logic Programming (CLP). Firstly, we show how CLP can be exploited to improve the software production process especially for retargetable code generation and test generation. CLP combines the declarative paradigm of logic programming with the efficiency of constraint solving techniques. CLP systems come with built-in mechanisms for solving constraints over various domains. For example, satisfiability checkers support Boolean constraints and IP-solvers support integer domains. Furthermore, CLP makes it easier to solve problems concurrently, e.g. the phase coupling problem during code generation.

Secondly, we present a solution for testing embedded processors. Thus we exploit CLP techniques for retargetable code generation to generate self-test programs, given a set of test patterns for each of the register transfer processor components.

1 INTRODUCTION

During the recent years, there has been a significant shift in the way complex electronic systems are implemented: various types of embedded processors are being used in many designs, which include off-the-shelf DSPs (e.g. TMS320C25 [27]), ASIPs [2] and in-house core processors. The advantages of these processors include: a very high flexibility in performing design changes and a short time-to-market. This shift in the implementation technology has largely been ignored by the scientific community, with the result that the tools for designing systems containing embedded processors are rather poor.

The situation is even worse when it comes to testing these systems. These systems are tested with ad hoc approaches, although it is well-known that processors can be tested systematically by running sophisticated test program diagnostics. Such test programs are used extensively for mainframe processors, but less so for embedded processors. Moreover, due to the high price of mainframes, it was acceptable to generate these test programs manually. For consumer products, this is no longer adequate and alternate, cost-effective ways of testing embedded processors have to be found. This chapter presents new techniques for testing embedded processors using the internal structure of the processor and exploiting retargetable compilation techniques to generate executable self-test programs.

2 RELATED WORK

As mentioned above, this chapter considers three different topics: Constraint Logic Programming, retargetable code generation and self-test program generation. To give the reader an impression of the context of this work, we briefly discuss some important works in these areas.

Systematic ways for testing microprocessors were first described by Abraham et al. [26, 7]. Their proposal relied on functional testing, i.e. it did neither require nor exploit knowledge about the internal structure of the processor to be tested. After some initial enthusiasm it was recognized that this resulted in a low efficiency and a poor coverage of real faults. Furthermore, this method was never integrated into a CAD system.

The interesting approach of Lee and Patel for testing microprocessors [17] uses the internal structure and a bidirectional discrete-relaxation technique, but does not aim at generating self-test programs.

This was different for the work on MSST by G. Krüger [15, 16]. Krüger exploited knowledge about the internal processor structure and consequently was able to generate more efficient test programs. MSST is a tool for hierarchical test generation: the user can specify test patterns for the processor components and MSST then produces executable programs generating these patterns and compares the response with a precomputed expected response.

MSST is possibly the first tool with the functionality described above, though its implementation has some severe limitations. It is implemented in an imperative language (Pascal) and thus suffers from the poor support of symbolic variables, automatic memory management and a low-level description style. Furthermore it is a large program and thus hard to maintain. Due to the above reasons, MSST cannot be adopted to new requirements (like the generation of external stimuli, variable instruction word lengths and support of multiple logic values).

Instead of incrementally trying to improve the situation, we came to the conclusion that the problems just mentioned are inherent in the traditional approach for implementing (CAD) software. Tools for VLSI CAD systems, commonly written in imperative languages, consist of a very large amount of source code. Maintenance, portability and adaptability are recurring problems. We realized that programming should proceed at a much higher level of abstraction and hence started to look at software technologies which provide a fundamentally different approach. We found **CLP** to be very well suited to our requirements.

Test program generation relies heavily on backtracking and the use of symbolic variables. Hence, logic programming languages such as Prolog provide a higher level of abstraction for implementing these tools. Thus, it was used by several researchers for this purpose [13, 25, 8], most of them concentrating on the gate level or even lower levels of abstraction. Unfortunately, the execution mechanism of standard Prolog results in a lot of backtracking and long execution times.

The situation is different for CLP languages [5], which became recently available (Prolog III [23], CHIP [9], ECLIPSE [10]). CLP systems come with built-in mechanisms for solving constraints over various domains. Satisfiability checkers support Boolean constraints and IP-solvers support integer domains. Hence, tools can be implemented at a higher level of abstraction. For example, it is possible to take advantage of the bidirectionality of clauses and simulate logic gates in both directions. In contrast to pure Prolog, no backtracking is required for forward simulation. Furthermore, several problems can be handled concurrently by specifying the subproblems with constraints and solve them in one step instead of solving subproblems sequentially. CLP languages have been used for test generation [24] for the gate level. Our work is the first one using CLP languages at the register transfer level.

It turns out that the techniques we propose can also be applied for retargetable code generation for general programming languages [12, 18, 29, 21, 22, 28, 11]. In fact, our techniques are capable of compiling a restricted set of programs into machine code.

3 RESTART: RETARGETABLE SELF-TEST PROGRAM GENERATION - AN OVERVIEW

RESTART is a retargetable self-test program generator which automatically **compiles** a user specified test program onto the given target processor. The result is an executable micro- or machine code and a set of external stimuli. It is intended to be used as internal off line tests, e.g. after a processor **restart**. RESTART (fig. 1) contains two inputs (processor description, test program specification) and two outputs (binary code, external stimuli).

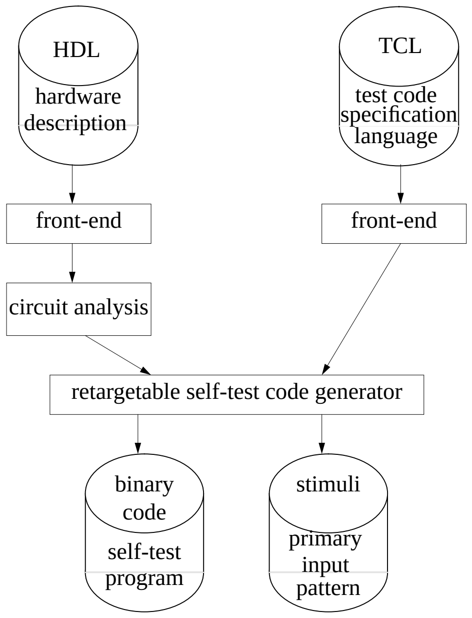

Figure 1 RESTART - System Overview

We use the description of the target architecture (hardware) and the test program specification (software) as **inputs**. The target architecture (processor) is expected to be described at the register transfer level either with VHDL [14] or MIMOLA [20].

TCL (Test program speCification Language) serves as a comfortable input language to specify self-test programs. A self-test program is specified by a test engineer, well acquainted with the RT structure of the processor. It is expected that a test engineer runs ATPG tools for each RT component to be tested, resulting in a set of test patterns for each RT component. Therefore, the fault coverage depends on the ATPG tool and the internal structure of the RT component to be tested. RESTART will achieve 100% fault coverage if the test patterns provided by the ATPG tool covers 100% of the component faults and if code generation is successful for all patterns. The test patterns are made available using TCL and RESTART generates binary code and stimuli which applies every test pattern to the RT components and checks the response. In this way, RESTART is a **hierarchical** test generation tool and RESTART based on single fault assumption, is independent of a special fault model. With our approach the obtained fault coverage depends on the TCL program. The human test engineer is responsible for: fault model, test strategy, fault coverage and test length. If specific hardware features for increasing the testability (e.g. a scan path) is available in the processor and described within the RT structure, this hardware can be used by the test code generator. A test of the controller is often a problem because of the low observability of the controller. In most cases, only an indirect test of the controller is possible.

The result of RESTART is an executable program and a set of external stimuli patterns. The program consists of a set of instructions. Each instruction is a pair (Label, BitString), i.e. an address within the instruction memory address range and a bit string consisting of 0, 1, X. A stimuli pattern is a triple (PrimaryInputName, Time, BitString). The time at which the bit string must stimulate the primary input is computed with respect to the clock cycle time of the processor. Primary inputs are expected to be justifyable without constraints. To validate the generated binary code an integrated simulator [6, 4] is able to simulate the circuit together with program and stimuli.

A summary of the main features of RESTART includes:

1. Optional compaction of the generated code.
2. Generation of external stimuli.
3. Test specification with a comfortable test specification language (TCL).
4. Declaration of an arbitrary number of variables in a register or memory component.
5. Concurrent application of transformation rules during resource allocation.
6. Concurrent and global scheduling, compaction and binding of the code.
7. Support for residual control.

The task of RESTART is to compile self-test programs. Compared to general programming languages, TCL is just a restricted language. RESTART exploits the special features of TCL programs to efficiently generate code for a wide range of architectures. Self-test programs contain a large amount of conditional jumps, comparison operations and constants (*'the test patterns'*) to be allocated. Therefore RESTART has knowledge about a set of transformation rules, e.g. for IF statements and comparison expressions. The special features of RESTART which are helpful to compile self-test programs are:

1. Compaction of the generated code is optional. The compaction phase can be switched off to simplify subsequent fault localization. If many instructions are executed in parallel, it could be more difficult to localize a fault.
2. Generation of external stimuli is possible, because the code generator must be able to allocate constants for all signals including primary inputs.
3. To deal with different hardware realizations for conditional jumps and comparison operations, a concurrent application of transformation rules during resource allocation is performed (i.e. code selection and resource allocation are coupled).
4. In order to allocate constants efficiently, potential constant sources and the paths from these sources to certain destinations are precomputed in a circuit analysis phase.

4 INPUT SPECIFICATION: HARDWARE AND SOFTWARE

4.1 Processor Description

For the specification of the target processor we use structural models. Datapath and controller must be completely described with MIMOLA or VHDL. Hardware descriptions must contain RT modules, their behavior and their interconnections. From this we generate an intermediate tree based format, representing the processor as a netlist of RT modules and the behavior of every RT module as a tree (see also fig. 6). The architectural assumptions under which a self-test program can be compiled are mentioned below. These assumptions must be made, in order to check if the output values of a component under test are as expected.

1. The processor must be able to perform a comparison operation (see fig. 7).
2. The processor must be able to perform a conditional jump.

DEMOCPU: A small Example Target Architecture

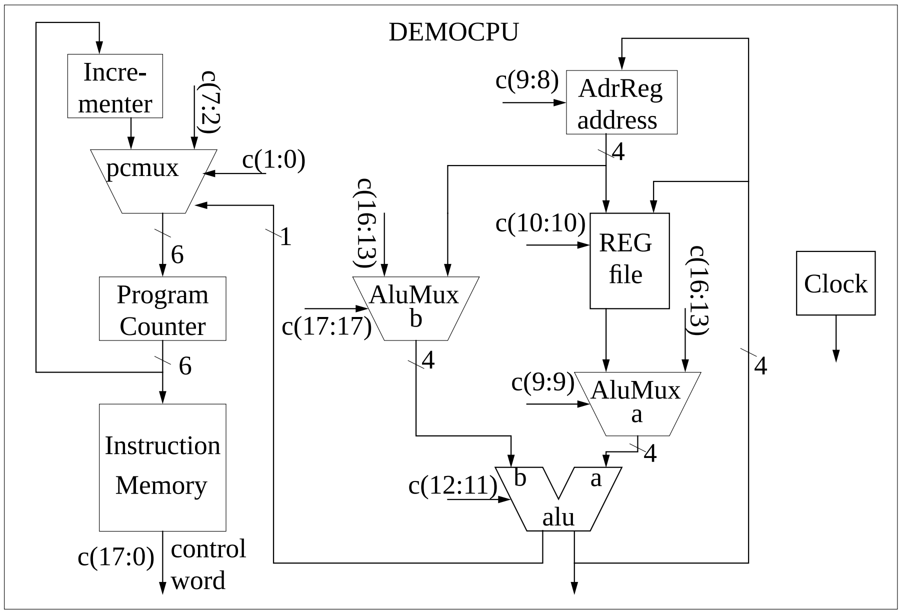

Figure 2 Example Target Processor

Fig. 2 shows a simple 4-bit processor consisting of 10 components. The datapath consists of a 16 x 4 register file, an address counter register, an alu and two multiplexers. A program counter, an instruction memory, an incrementer and a multiplexer make up the controller. All registers are synchronized by a clock. Control signals are denoted by c followed by an index range (high-bit:low-bit). The 6-bit program counter addresses the 64 x 18 bit instruction memory. The alu is specified in VHDL as shown in fig. 3, with a condition output that enables the controller multiplexer to perform conditional jumps. DEMOCPU serves as a running example throughout the chapter.

```
ENTITY alu IS PORT
 (a, b  :  IN  bit_vector (3 Downto 0);
  ctr   :  IN  bit_vector (1 Downto 0);
  result:  OUT bit_vector (3 Downto 0); condition:  OUT bit);
END alu;
ARCHITECTURE behavior OF alu IS BEGIN
 WITH ctr SELECT result    <=
            a                      WHEN "00",
            b                      WHEN "01",
            a+b                    WHEN "10",
            a-b                    WHEN "11";
 WITH ctr SELECT condition <=
            bool2bit(a = 0)        WHEN "00",
            bool2bit(b = 0)        WHEN "01",
            bool2bit((a + b) = 0) WHEN "10",
            bool2bit((a - b) = 0) WHEN "11";
END behavior;
```

Figure 3 DEMOCPU alu

4.2 TCL

TCL is an imperative language in which the following kinds of test statements can be used by a test engineer to specify a self-test program (# precedes a hexadecimal number; % precedes a binary number; a variable location is referred to by $\langle ComponentName\rangle/\langle VariableName\rangle$; all examples are related to DEMOCPU, see fig. 2):

- An **Initialization** causes the test generator to produce code for loading a register or one cell of a memory with a constant initialization value. Examples: *REG/file[1] := #A; AdrReg/addr := 6;*
- A **Read Test** makes the test generator produce code for testing, if a memory cell or a register contains a certain value. Examples: *TEST REG/file[1] = #A; TEST AdrReg/addr = 6;*
- An **Initialization and Read Test** combines an initialization with a read test, i.e. the generated code first loads the specified location with a value

and then checks, if it really contains that value.
Examples: *TEST REG/file[1] := #A; TEST AdrReg/addr := 6;*

- A **Component Test** makes the test generator produce code for testing the functionality of any module, i.e. the related module's input ports are stimulated with the specified values, and then the outputs are checked for correctness. The programmer has only to specify the input values and an integrated structure simulator calculates the corresponding output values. An underscore may be used to denote a port of the module which is not relevant to the test whereas X denotes a binary don't care.
Example: *TEST alu(#A,#F,%11);*
- A **Loop** is used to apply one of the first four kinds of statements several times with one argument iterating over a range of values. Examples:
FOR adr := 1 TO 15 DO TEST REG/file[adr] := #A;
FOR ctr := 0 TO 3 DO TEST alu(#5,#F,ctr);

The meaning of the keyword TEST is the following: RESTART is directed to generate code that checks if the output ports of a certain component are as expected. Therefore a conditional jump is generated:

```
IF component answer = expected answer
    THEN increment program counter ELSE jump to error label;
```

If no error occurs, the program continues with the execution of the next instruction of the self-test program, otherwise a jump to an error procedure is performed. TCL allows the specification of all kinds of tests including memory test loops. A TCL program is a sequence of TCL statements.

5 RETARGETABLE CODE GENERATION: TECHNIQUES

5.1 Circuit Analysis

In the circuit analysis phase (fig. 1), the given processor is analyzed and a subset of the instruction set is extracted. The result is a list of microoperations the processor can perform and contains: register transfer moves, conditional and unconditional jumps, counter increment operations, etc. The considered subset is powerful enough to deal with the compilation of TCL programs as described above.

Microoperations are stored as facts (a fact with arity n and functor f is a term $f(x_1, \ldots, x_n)$), e.g. transparent(alu, (result,%0000,%10), (result,condition)) considers the fact, that the DEMOCPU alu is able to switch the input a to the output result by adding a to the neutral element %0000 (add operation selected by the binary control code c(12:11) = %10).

An important subtask of code generation, is to generate code for unconditional jumps, i.e. to move a constant value into the program counter without consideration of a condition from the datapath. DEMOCPU has only one possibility (see also fig. 6) to perform such an unconditional jump by selecting c(1:0) = %01 as control code for the multiplexer pcmux. Therefore, the fact jump([(instructionmemory,(_)), (pcmux,(%X,%01,_JUMPADR)), (programcounter,(JUMPADR,%X))]) simply denotes a path from the instruction memory (source) to the program counter (destination). JUMPADR is the symbolic jump address. For every component of the path the values of the input ports are precomputed.

5.2 Retargetable Code Generation

Fig. 4 shows the program flow of the retargetable code generator. A hardware description, the output of the circuit analysis phase and the TCL program serve as inputs. The code generation phase described in the next subsection computes a relocatable program. With respect to a certain program counter initialization value, the relocatable program has to be scheduled and linked to a designated program start address. The retargetable self-test code generator is able to compact the generated code optionally. To allow a) detailed analyses of the hardware and b) subsequent fault localization, it must be possible to switch off the compaction phase. The user is asked if the code should be compacted or left uncompacted. Finally unused registers, memories and tristate bus drivers must be disabled and the instructions are composed to complete control store words by adding a program counter increment or jump operation (with respect to the realization of the controller). An absolute program and a set of external stimuli is the result.

Code Generation

The task of the code generator is to map a sequence of TCL statements onto the hardware. Each TCL statement is decomposed into a set of **simple instructions** consisting of assignments and conditional jumps. The main idea of the code generation algorithm is as follows:

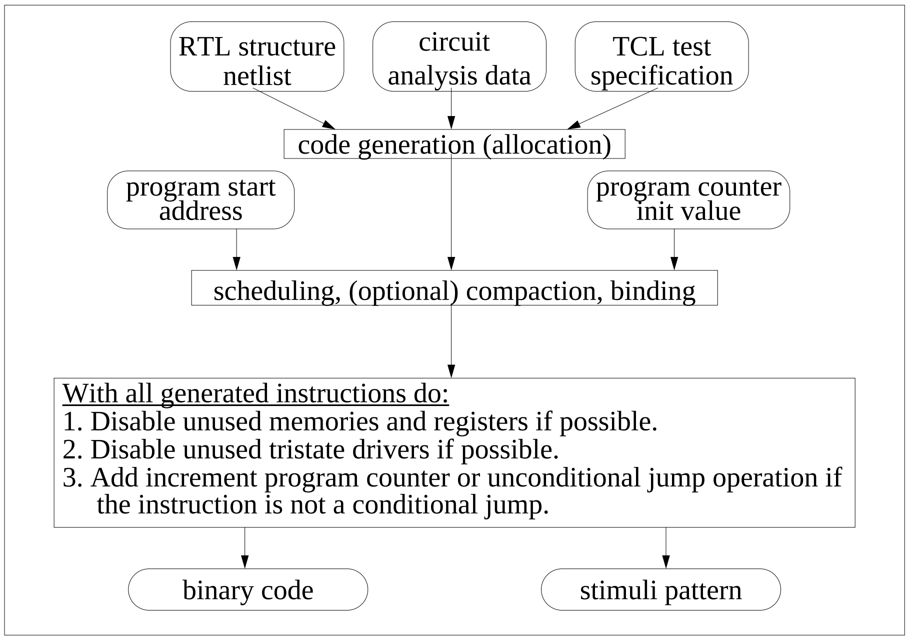
RTL structure netlist
circuit analysis data
TCL test specification
code generation (allocation)
program start address
program counter init value
scheduling, (optional) compaction, binding
With all generated instructions do:
1. Disable unused memories and registers if possible.
2. Disable unused tristate drivers if possible.
3. Add increment program counter or unconditional jump operation if the instruction is not a conditional jump.
binary code
stimuli pattern

Figure 4 Program Flow

1. A simple instruction can be represented as a tree.
2. The behavior of every RT component can be represented as a tree.
3. Retargetable compilation means: Mapping of a sequence of simple instruction trees to a netlist where each node consists of a behavior tree of an RT component.

EXAMPLE: Assume, the following conditional jump statement has to be compiled onto DEMOCPU:

IF condition THEN increment program counter ELSE jump to label;

Fig. 5 shows the tree representation of above conditional jump statement. Fig. 6 shows the view of the corresponding part of the hardware. IF statements are nested in a CASE construct to allow a conditional selection of one of two input branches. To compile the conditional jump statement, an **allocation** routine has to search for a multiplexer (i.e. a (sub-) tree as shown in Fig. 5), starting from the destination (program counter) backwards through the circuit to the sources (condition, program counter, instruction memory). The resulting instruction justifies the control input of pcmux with 2 (binary %10) and loads the program counter.

To deal with different target architectures, different alternatives to map simple instruction trees on RT behavior trees must be taken into account. This is done by **transformation rules**. For instance a statement **X := Y+1** can be transformed to **X := increment(Y)**. A comparison operation, as needed for the TEST statement, **(component answer = expected answer)** can be transformed to **((component answer - expected answer) = 0)**. Loops are transformed into a sequence of simple instructions. To represent transformation rules for simple instructions we use **structural constraints** implemented in CLP. Consider the following definitions:

Definition 1.1 (Constraint) *Let $V = \{X_1, \ldots, X_n\}$ be a finite set of variables, which take their values from their finite domains $D_1, \ldots, D_n$. A* constraint *$c(X_{i_1}, \ldots, X_{i_k})$ between k variables from V is a subset of the Cartesian Product $D_{i_1} \times \ldots \times D_{i_k}$.*

The domain of variables within structural constraints is the set of trees, whereas the domain of variables within linear constraints is the set of integer numbers.

Definition 1.2 (Transformation Rule) *Let X_1, X_2 be two variables, both variables representing a tree. A* transformation rule for a simple statement *is a structural constraint $tr(X_1, X_2)$.*

The meaning is: A tree X_1 can be transformed to a tree X_2 if $tr(X_1,X_2)$ is true.

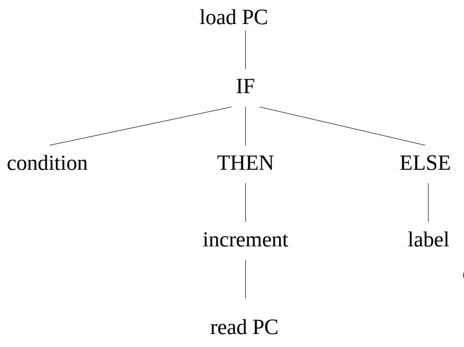

Figure 5 Tree representation of a conditional jump statement

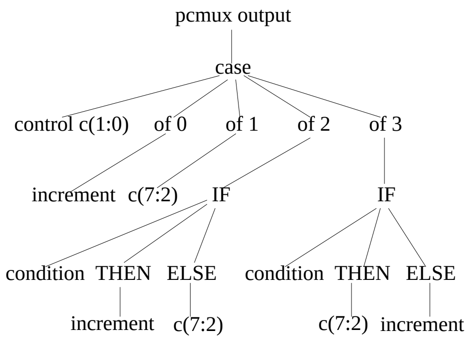

Figure 6 Behavior tree of the multiplexer pcmux

Example: Let X_1 be a comparison operation (A=B). Then X_1 can be transformed to trees X_2 as shown in fig. 7. Of course there exists further trees into which X_1 can be transformed, e.g. commutativity can be exploited by exchanging the sons of a commutative operator.

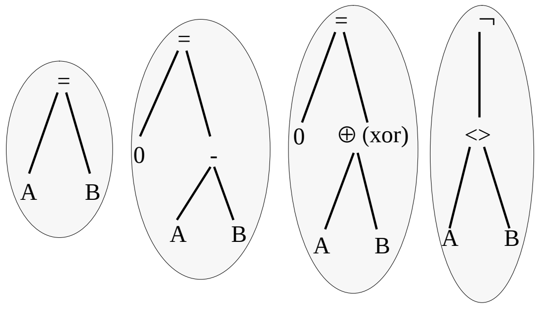

Figure 7 Transformations for a Comparison Operation

Allocation: We now describe the allocation of a simple instruction. In contrast to most previous retargetable compilers, allocation and application of transformation rules can be done concurrently within a CLP system. Therefore a variable, representing a simple instruction which has to be allocated in the circuit, is constrained to a set of alternative trees. Allocation starts at the destination (e.g. the left hand side of an assignment) and from there a recursive search backward through the circuit is performed as follows:

allocate(statement tree, destination)
The predecessor RT component of the destination is determined and the following cases are distinguished:

1. The statement tree can be mapped onto the predecessor behavior tree: success
2. The predecessor is a register or memory: insert a new control step and use the predecessor as temporary cell; call allocate(statement tree, predecessor)
3. A subtree including the root of the statement tree can be mapped onto the predecessor behavior tree: call allocate('rest' of statement tree, predecessor)
4. The output of the predecessor can be switched to an input (transparent mode): call allocate(statement tree, predecessor)

5. otherwise: fail

Steps 1. and 4. allow the application of a transformation rule. During allocation, components currently under test, are locked to avoid failure masking, i.e. no data transfers through these components are permitted. Allocation of constants terminates at components allowed as **constant sources**: instruction memory, primary inputs and decoders. Due to lack of space, the complexity of the allocation phase is not discussed. Constant allocation in general is NP-complete. The allocation result is a relocatable program and a set of constraints representing data dependencies, dependencies between addresses etc.

Example: Assume, the statement REG/file[1] := #F; has to be compiled on DEMOCPU. Therefore, two constants have to be allocated: address (%0001) and data (%1111). To allocate data, the predecessor component of the register file is determined: the alu output result. Because the constant %1111 can not be mapped to the alu the algorithm tries to switch the alu in a transparent mode. This is done by setting c(12:11) to %00. A recursive search for input a of the alu is started. Control signal c(9:9) is set to %0 and finally the constant allocation terminates at the instruction memory by setting c(16:13) to %1111. To allocate the address, the address register has to be loaded in a previous instruction (case 2 of above allocation procedure). Afterwards, the constant %0001 is allocated using component AluMuxb.

Note, for efficiency reasons the allocation of constants is accelerated by using the precomputed facts from the circuit analysis phase. Consequently, the above allocation of the constant %1111 is just a move instruction (path) from the instruction memory to the register file and is computed without backtracking. Some heuristics are used during allocation. E.g. operations with a smaller number of necessary constants are preferred.

Table 1 shows the resulting relocatable program. Additionally the constraint $L_1 < L_2$ has been generated. Note, till now no increment operation for the program counter has been generated (c(1:0) is unbound).

Label	17	16:13	12:11	10	9:8	7:2	1:0	Comment
L_1	1	0001	01	1	10	XXXXXX	XX	AdrReg := 1
L_2	X	1111	00	0	00	XXXXXX	XX	REG[1]:=#F

Table 1 Relocatable Program

Scheduling, Compaction, Binding

After code generation, a relocatable program consisting of a set of instructions and a set of partially ordered labels is generated. Therefore, three tasks have to be done: A program has to be scheduled, linked and optionally compacted. Every label has to be bound to a number within the address range of the instruction memory and a total order of the labels and the corresponding instructions has to be found. Relocatable code is mapped to absolute code. Instructions which can be executed in parallel can be compacted, i.e. two or more instructions are merged to one instruction.

We perform **global scheduling** while concurrently compacting and binding the code. Here we make extensive use of linear constraints over the integer domain. In this way it is possible to exploit the parallelism of the target processor. Global scheduling is possible because of the specific structure of the basic blocks of self-test programs, mainly consisting of move and conditional jump statements. A (simplified) formal description of the scheduling, compaction and binding phase follows. First we distinguish between absolute code and relocatable code. Thereafter, we define what kind of constraints are allowed to represent dependencies between variables and labels. Next, we define necessary preconditions to merge two instructions. An example illustrates, how instructions are merged together. Let Start, Address, End and n be natural numbers. Start $\leq$ Address $\leq$ End, is the address range of the instruction memory and n its width.

Definition 1.3 (Relocatable Code) *Let L be a set of labels and V be a set of variables.* Relocatable Code RC *is a tuple $RC = (P, C)$ with $P = \{(L_i, I_i) | L_i \in L, I_i \in \{\{0, 1, X\} \cup V\}^n\}$ and C is a set of linear constraints over $L \cup V$.*

The set V is used to represent dependencies between the instructions and the labels. For instance jump addresses usually are coded within the instructions and every variable $V_i \in V$ finally represents a binary number.

Definition 1.4 (Absolute Code) Absolute Code AC *is a set of tuples $AC = \{(L_i, I_i) | Start \leq L_i \leq End, I_i \in \{0, 1, X\}^n\}$, i.e. L_i is a bound label and I_i is the corresponding instruction.*

Let **P(I, k)** be the **projection** of a bit string on the k-th bit (high-bit on the left, low-bit on the right of a bit string. The rightmost bit position is 0).

Definition 1.5 (compatible) *Assuming* $I_i, I_j \in \{\{0,1,X\} \cup V\}^n$ *are relocatable instructions. The predicate* compatible(I_i, I_j) *is true iff:* $\forall k, 0 \leq k \leq n-1:$

$$(P(I_i,k) = P(I_j,k)) \vee (P(I_i,k) = X) \vee (P(I_j,k) = X) \vee$$
$$(P(I_i,k) \in V \wedge P(I_j,k) \notin V) \vee (P(I_j,k) \in V \wedge P(I_i,k) \notin V)$$

If compatible(I_i, I_j) is true, we say I_i and I_j are compatible. Instructions which are compatible are candidates to be compacted. With above formalism, scheduling, compaction and binding is reduced to the problem of solving a system of linear equations and inequalities.

Example: The following TCL program is an extension of the previous example:

```
REG/file[1] := #F;          – load register cell 1
PCREG/pc := 10;             – jump to 10
TEST AdrReg/addr = 1;       – check if address register content is 1
```

Label	17	16:13	12:11	10	9:8	7:2	1:0	Comment
L_1	1	0001	01	X	10	XXXXXX	XX	AdrReg := 1
L_2	X	1111	00	0	0X	XXXXXX	XX	REG[1] :=#F
L_3	X	XXXX	XX	X	XX	ABDEFG	XX	PC:=10
L_4	0	0001	11	X	0X	HIJKLM	10	check address

Table 2 Relocatable Program P

After allocation the relocatable program RC = (P,C) as shown in table 2 is generated. The set of constraints is C = $\{L_1 < L_2, L_2 \leq L_3, L_3 \leq L_4, L_1 = 6, L_4 = 10, G{+}2F{+}4E{+}8D{+}16B{+}32A = L_4, M{+}2L{+}4K{+}8J{+}16I{+}32H = ErrorLabel, \forall V_i \in \{A,B,D,E,F,G,H,I,J,K,L,M\} : V_i \in \{0,1\}\}$.

RC can be mapped to the absolute code AC given in table 3 (A = address). Note, RC has been linked to the (selected) constant program start address 6. Additionally, RC has been composed to complete control store words. Therefore e.g. a jump instruction has been merged to the instruction with address 7 by setting c(1:0) = %10. Instructions L_2 and L_3 have been compacted. Some unused registers have been disabled at certain control steps (e.g. the register file by setting c(10) = %1). The error label (c(7:2) = 63) is given by the user.

A	17	16:13	12:11	10	9:8	7:2	1:0	Comment
6	1	0001	01	1	10	X···X	00	AdrReg := 1
7	X	1111	00	0	00	001010	01	REG[1]:=#F; PC:=10
10	0	0001	11	1	00	111111	10	check address

Table 3 Absolute Code for DEMOCPU

Above formalism is flexible and powerful enough to handle complex address restrictions. Linear constraints are general enough to express strange address generation schemes (even the ones described in [3]). Scheduling, compaction and binding can be handled concurrently and with a minimum of programming effort (the complete scheduling, compaction and binding phase has about 200 lines of code!) using the built-in constraint solving mechanism for the integer domain and the Prolog inherent backtracking mechanism.

6 RESULTS

A retargetable self-test code generator (6500 lines of code) has been fully implemented in the constraint logic programming language ECLIPSE [10]. Half of these lines of code are comments and so CLP programs are pretty short compared to imperative implementations (ratio ~1:4). We applied the system to a variety of digital processors to show the efficiency of the new techniques. The results shown here, indicate that an implementation with CLP can be applied to realistic structures.

Circuit	RTL modules	instruction memory width	datapath width
simplecpu	10	20	4
democpu	11	18	4
demo	16	84	16
prips	50	83	32
mano	21	50	16

Table 4 Example Processors Circuit Information

Table 4 describes the example circuits: the general purpose microprocessors simplecpu [6], demo [20], mano [19] and DEMOCPU (fig. 2); prips [1] is a coprocessor with a RISC-like instruction set, which provides data types and instructions supporting the execution of Prolog programs. The number of RTL components, the width of the datapath and the width of the microinstruction controller is given.

			uncompacted			compacted		
Circuit	#TCL	#S	#μI	sec	#μI/s	#μI	sec	#μI/s
simplecpu	7	5	11	0.71	15.5	11	0.71	15.5
democpu	5	0	20	1.09	18.3	18	1.15	15.6
demo	17	73	102	26.1	3.9	91	26.46	3.4
prips	7	0	17	20.2	0.84	17	20.5	0.83
mano	15	1	136	37.41	3.63	113	36.5	3.1

Table 5 RESTART Results

Table 5 shows the results for the retargetable self-test program generator. The number of compiled TCL instructions (note, even a memory test loop is only one TCL instruction), the generated number of stimuli patterns (#S), the number of generated instructions (#μI), CPU time in seconds and the ratio (generated instructions per second) is given. All times are measured on a SPARC 20 workstation. The results for code generation without compaction and the results for programs which have been compacted are given. It can be seen, that the CPU times for both cases are very similar because a) the compaction is done very fast and b) the saved time is consumed by the output handling of more instructions. These TCL programs just serve to demonstrate the compilation speed. All generated programs have been validated with the above mentioned simulator. A small number of primary input stimuli patterns indicates, that the processor is mainly able to test itself, whereas a large amount of stimuli patterns indicates that certain constants can not be allocated within the circuit. Compaction of self-test programs only results in 10% - 20% less code because test programs usually are not highly parallel.

One of the 5 TCL instructions for democpu is a test loop for detecting faults in the instruction decoding and control function of AluMuxa: FOR ctr := 0 TO 1 DO TEST alumuxa(#A,#C,ctr); This test loop has been compiled by RESTART and the resulting self-test program has been stored as initialization for the microinstruction memory. Now we slightly modified the hardware description of the multiplexer, i.e. we modified the instruction decoding and control function of the multiplexer resulting in a "faulty" multiplexer. The rest of the processor has been left unchanged. An RT simulation of the "faulty" processor together with the self-test program has been performed and of course the injected fault has been detected.

7 CONCLUSIONS

We have shown that test programs for embedded processors can be automatically generated. The generation process essentially consists of matching a test code specification against a structural description of the processor. This process has been viewed as a special case of retargetable code generation. It has been possible to compile self-test programs for several processors.

Furthermore, we have shown how the built-in support for symbolic variables and constraints over these can lead to a more efficient software production process. Several subproblems can be handled concurrently. It is well known that the consideration of all relevant design constraints is a key issue in CAD. CLP languages have built-in mechanisms for such constraints and we have successfully exploited the potential that is inherent in one of these languages.

Acknowledgements

I gratefully acknowledge the work that has been done by Andreas Neumann during the recent years to bring this project to an successful end. Furthermore I wish to thank Peter Marwedel for fruitful discussions and comments. This work was supported by the DFG, the German research foundation.

REFERENCES

[1] C. Albrecht, S. Bashford, P. Marwedel, A. Neumann, W. Schenk. "The Design of the PRIPS Microprocessor", 4th EUROCHIP-Workshop on VLSI Training, Toledo - Spain, September 1993, pp. 254-259.

[2] A. Alomary, T. Nakata, Y. Honma, M. Imai, N. Hikichi "An ASIP instruction set optimization algorithm with functional module sharing cons-

traint", Int. Conf. on Computer-Aided Design (ICCAD), Santa Clara, November 1993, pp. 526-532.

[3] T. Baba, H. Hagiwara. "The MPG System: A Machine-Independent Efficient Microprogram Generator", IEEE Trans. on Computers, Vol. C-30, June 1981, pp. 373-395.

[4] R. Beckmann, U. Bieker, I. Markhof. "Application of Constraint Logic Programming for VLSI CAD Tools", First Int. Conference Constraints in Computational Logic, Munich, September 1994, LNCS 845, pp. 183-200.

[5] F. Benhamou, A. Colmerauer (editors). "Constraint Logic Programming: Selected Research", Cambridge, MA: MIT Press, 1993.

[6] U. Bieker, A. Neumann. "Using Logic programming and Coroutining for Electronic CAD", 2nd Int. Conf. on the Practical Applications of Prolog, London, April 1994, pp. 67-78.

[7] D. Brahme, J. A. Abraham. "Functional Testing of Microprocessors", IEEE Transactions on Computers, Vol. C-33, No. 6, June 1984, pp. 475-485.

[8] W. F. Clocksin. "Logic Programming and Digital Circuit Analysis", The Journal of Logic Programming, March 1987, pp. 59 - 82.

[9] "CHIP User's Guide", COSYTEC SA, Parc Club Orsay Universite, 4, rue Jean Rostand, 91893 Orsay Cedex, France, 1991.

[10] "ECLIPSE 3.4 User Manual", ECRC Common Logic Programming System. ECRC GmbH, Arabellastr. 17, Munich, Germany, 1994.

[11] A. Fauth, A. Knoll. "Automated generation of DSP program development tools using a machine description formalism", Int. Conf. on Audio, Speech and Signal Processing, 1993.

[12] M. Ganapathi, C.N. Fisher, J.L. Henessy. "Retargetable Compiler Code Generation", ACM Computing Surveys, Vol. 14, (4) 1982.

[13] P. W. Horstmann. "Automation of the Design for Testability Using Logic Programming", Dissertation, University of Missouri, October 1983.

[14] Design Automation Standards Subcommittee of the IEEE. "Draft standard VHDL language reference manual", IEEE Standards Department, 1992.

[15] G. Krüger. "Automatic Generation of Self-Test Programs - A New Feature of the MIMOLA Design System", 23rd Design Automation Conference, Las Vegas, June 1986, pp. 378-384.

[16] G. Krüger. "A Tool for Hierarchical Test Generation", IEEE Transactions on Computer Aided Design of Integrated Circuits and Systems, Vol. 10, No. 4, April 1991, pp. 519-524.

[17] J. Lee, J. Patel. "An Instruction Sequence Assembling Methodology for Testing Microprocessors", International Test Conference, Baltimore, September 1992, pp. 49-58.

[18] C. Liem, P. Paulin. "Flexware - A flexible firmware development environment", Proc. European Design & Test Conference, March 1994, pp. 31-37.

[19] M. Morris Mano. "Computer System Architecture", Prentice-Hall Int., Inc., Third Edition, 1993.

[20] S. Bashford, U. Bieker, B. Harking, R. Leupers, P. Marwedel, A. Neumann, D. Voggenauer. "The MIMOLA Language - Version 4.1", Technical Report, Computer Science Dpt., University of Dortmund, September 1994.

[21] L. Nowak, P. Marwedel. Verification of Hardware Descriptions by Retargetable Code Generation. 26th Design Automation Conference, Las Vegas, June 1989, pp. 441-447.

[22] J. V. Praet, G. Goossens, D. Lanneer, H. D. Man. "Instruction Set Definition and Instruction Selection for ASIPs", 7.th Int. Symposium on High-Level Synthesis, Canada, May 1994, pp. 11-16.

[23] "Prolog III Reference Manual", PrologIA, Parc Technologique de Luminy - Case 919, 13288 Marseille Cedex 09, France, 1991.

[24] H. Simonis. "Test Generation using the Constraint Logic Programming Language CHIP", In Proceedings of the 6th International Conference on Logic Programming, Lisboa, Portugal, June 1989, pp. 101-112.

[25] D. Svanaes, E. J. Aas. "Test generation through logic programming", North-Holland, INTEGRATION, the VLSI journal, February 1984, pp. 49-67.

[26] S. M. Thatte, J. A. Abraham. "Test Generation for Microprocessors", IEEE Transactions on Computers, Vol. C-29, No. 6, June 1980, pp. 429-441.

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

[28] B. Wess. "Optimizing Signal Flow Graph Compilers For Digital Signal Processors", Proc. ICSPAT, 1994.

[29] T. Wilson, G. Grewal, B. Halley, D. Banerji. “An Integrated Approach to Retargetable Code Generation”, 7. th Int. Symposium on High-Level Synthesis, Canada, May 1994, pp. 70-75.