# Retargetable compilers and architecture exploration for embedded processors

R. Leupers, M. Hohenauer, J. Ceng, H. Scharwaechter, H. Meyr, G. Ascheid and G. Braun

**Abstract:** Retargetable compilers can generate assembly code for a variety of different target processor architectures. Owing to their use in the design of application-specific embedded processors, they bridge the gap between the traditionally separate disciplines of compiler construction and electronic design automation. In particular, they assist in architecture exploration for tailoring processors towards a certain application domain. The paper reviews the state-of-the-art in retargetable compilers for embedded processors. Based on some essential compiler background, several representative retargetable compiler systems are discussed, while also outlining their use in iterative, profiling-based architecture exploration. The LISATek C compiler is presented as a detailed case study, and promising areas of future work are proposed.

## 1 Introduction

Compilers translate high-level language source code into machine-specific assembly code. For this task, any compiler uses a model of the target processor. This model captures the compiler-relevant machine resources, including the instruction set, register files and instruction scheduling constraints. While in traditional target-specific compilers this model is built-in (i.e. it is hard-coded and probably distributed within the compiler source code), a *retargetable compiler* uses an external processor model as an additional input that can be edited without the need to modify the compiler source code itself (Fig. 1). This concept provides retargetable compilers with high flexibility with respect to the target processor.

Retargetable compilers have been recognised as important tools in the context of embedded system-on-chip (SoC) design for several years. One reason is the trend towards increasing use of programmable processor cores as SoC platform building blocks, which provide the necessary flexibility for fast adoption, e.g. of new media encoding or protocol standards and easy (software-based) product upgrading and debugging. While assembly language used to be predominant in embedded processor programming for quite some time, the increasing complexity of embedded application code now makes the use of high-level languages like C and C++ just as inevitable as in desktop application programming.

In contrast to desktop computers, embedded SoCs have to meet very high efficiency requirements in terms of MIPS per Watt, which makes the use of power-hungry, high-performance off-the-shelf processors from the desktop computer domain (together with their well developed compiler technology) impossible for many applications. As a consequence, hundreds of different domain or even application-specific programmable processors have appeared in the semiconductor market, and this trend is expected to continue. Prominent examples include low-cost/low-energy microcontrollers (e.g. for wireless sensor networks), number-crunching digital signal processors (e.g. for audio and video codecs), as well as network processors (e.g. for internet traffic management).

All these devices demand for their own programming environment, obviously including a high-level language (mostly ANSI C) compiler. This requires the capability of quickly designing compilers for new processors, or variations of existing ones, without the need to start from scratch each time. While compiler design traditionally has been considered a very tedious and manpower-intensive task, contemporary retargetable compiler technology makes it possible to build operational (not heavily optimising) C compilers within a few weeks and more decent ones approximately within a single man-year. Naturally, the exact effort heavily depends on the complexity of the target processor, the required code optimisation and robustness level, and the engineering skills. However, compiler construction for new embedded processors is now certainly much more feasible than a decade ago. This permits us to employ compilers not only for application code development, but also for optimising an embedded processor architecture itself, leading to a true 'compiler/ architecture codesign' technology that helps to avoid hardware–software mismatches long before silicon fabrication.

This paper summarises the state-of-the-art in retargetable compilers for embedded processors and outlines their design and use by means of examples and case studies. We provide some compiler construction background needed to understand the different retargeting technologies and give an overview of some existing retargetable compiler systems. We describe how the above-mentioned 'compiler/architec-architecture codesign' concept can be implemented in a processor architecture exploration environment. A detailed example of an industrial retargetable C compiler system is discussed.

© IEE, 2005
*IEE Proceedings* online no. 20045075
doi: 10.1049/ip-cdt:20045075
Paper first received 7th July and in revised form 27th October 2004

R. Leupers, M. Hohenauer, J. Ceng, H. Scharwaechter, H. Meyr, and G. Ascheid are with the Institute for Integrated Signal Processing Systems, RWTH Aachen University of Technology, SSS-611920, Templergraben 55, D-52056, Aachen, Germany
G. Braun is with CoWare Inc., CA 95131, USA
E-mail: leupers@iss.rwth-aachen.de

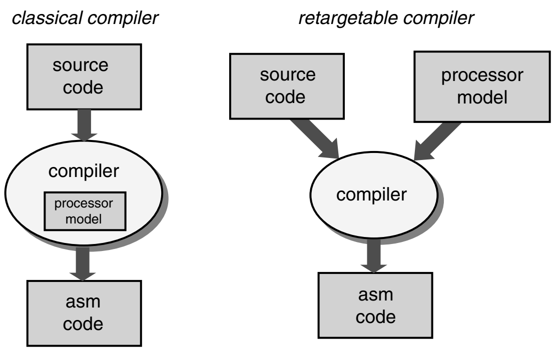


**Fig. 1** *Classical against retargetable compiler*

## 2 Compiler construction background

The general structure of retargetable compilers follows that of well proven classical compiler technology, which is described in textbooks such as [1–4]. First, there is a language frontend for source code analysis. The frontend produces an intermediate representation by which a number of machine-independent code optimisations are performed. Finally, the backend translates the intermediate representation into assembly code, while performing additional machine-specific code optimisations.

### 2.1 Source language frontend

The standard organisation of a frontend comprises a scanner, a parser and a semantic analyser (Fig. 2). The scanner performs lexical analysis on the input source file, which is first considered just as a stream of ASCII characters. During lexical analysis, the scanner forms substrings of the input string to groups (represented by *tokens*), each of which corresponds to a primitive syntactic entity of the source language, e.g. identifiers, numerical constants, keywords, or operators. These entities can be represented by regular expressions, for which in turn finite automata can be constructed and implemented that accept the formal languages generated by the regular expressions. Scanner implementation is strongly facilitated by tools like lex [5].

The scanner passes the tokenised input file to the parser, which performs syntax analysis with respect to the context-free grammar underlying the source language. The parser recognises syntax errors and, in the case of a correct input, builds up a tree data structure that represents the syntactic structure of the input program.

Parsers can be constructed manually based on the LL(k) and LR(k) theory [2]. An LL(k) parser is a top-down parser, i.e. it tries to generate a derivation of the input program from the grammar start symbol according to the grammar rules. In each step, it replaces a non-terminal by the right-hand side of a grammar rule. In order to decide which rule to apply out of possibly many alternatives, it uses a lookahead of *k* symbols on the input token stream. If the context-free grammar shows certain properties, this selection is unique, so that the parser can complete its job in linear time in the input size. However, the same also holds for LR(k) parsers which can process a broader range of context-free grammars. They work bottom-up, i.e. the input token stream is reduced step-by-step until finally reaching the start symbol. Instead of making a reduction step solely based on the knowledge of the *k* lookahead symbols, the parser additionally stores input symbols temporarily on a stack until enough symbols for an entire right-hand side of a grammar rule have been read. Due to this, the implementation of an LR(k) parser is less intuitive and requires more effort than for LL(k).

Constructing LL(k) and LR(k) parsers manually provides some advantage in parsing speed. However, in most practical cases tools like yacc [5] (that generates a variant of LR(k) parsers) are employed for semi-automatic parser implementation.

Finally, the semantic analyser performs correctness checks not covered by syntax analysis, e.g. forward declaration of identifiers and type compatibility of operands. It also builds up a symbol table that stores identifier information and visibility scopes. In contrast to scanners and parsers, there are no widespread standard tools like lex and yacc for generating semantic analysers. Frequently, attribute grammars [1] are used, though, for capturing the semantic actions in a syntax-directed fashion, and special tools like ox [6] can extend lex and yacc to handle attribute grammars.

### 2.2 Intermediate representation and optimisation

In most cases, the output of the frontend is an intermediate representation (IR) of the source code that represents the input program as assembly-like, yet machine-independent low-level code. Three-address code (Figs. 3 and 4) is a common IR format.

There is no standard format for three-address code, but usually all high-level control flow constructs and complex expressions are decomposed into simple statement sequences consisting of three-operand assignments and gotos. The IR generator inserts temporary variables to store intermediate results of computations.

Three-address code is a suitable format for performing different types of flow analysis, i.e. control and data flow analysis. Control flow analysis first identifies the basic block structure of the IR and detects the possible control transfers between basic blocks. (A basic block is a sequence of IR statements with unique control flow entry and exit points). The results are captured in a control flow graph (CFG). Based on the CFG, more advanced control flow analyses can be performed, e.g. in order to identify program loops. Figure 5 shows the CFG generated for the example from Figs. 3 and 4.

Data flow analysis works on the statement level and determines interdependencies between computations.

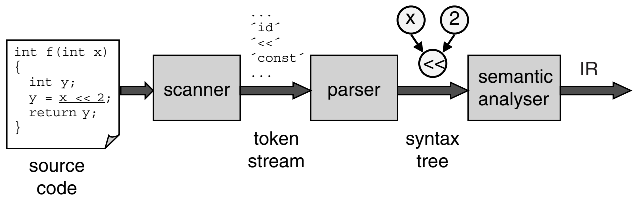

```
int f(int x)
{
  int y;
  y = x << 2;
  return y;
}
```

**Fig. 2** *Source language frontend structure*

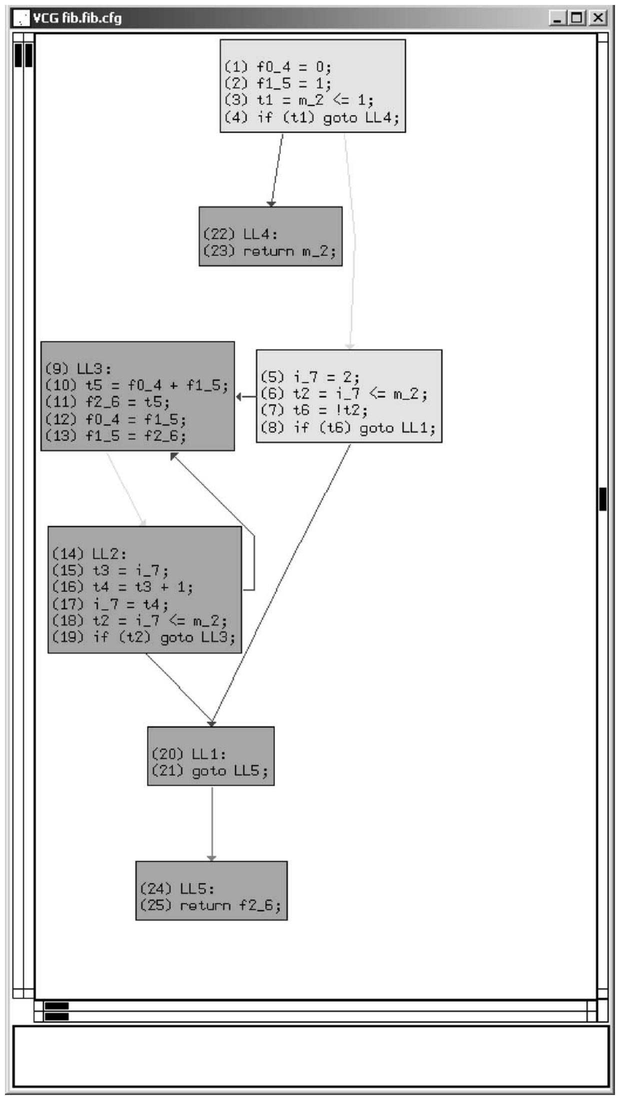
```
int fib (int m)
{  int f0 = 0, f1 = 1, f2, i;
   if (m <= 1) return m;
   else
   for (i = 2; i <= m; i++) {
     f2 = f0 + f1;
     f0 = f1;
     f1 = f2; }
   return f2;
}
```

**Fig. 3** *Sample C source file fib.c (Fibonacci numbers)*

```
int fib (int m2)
{
 int f0_4, f1_5, f2_6, i_7, t1, t2, t3, t4, t6, t5;

          f0_4 = 0;
          f1_5 = 1;
          t1 = m_2 <= 1;
          if (t1) goto LL4;
          i_7 = 2;
          t2 = i_7 <= m_2;
          t6 = ! t2;
          if (t6) goto LL1;
 LL3:     t5 = f0_4 + f_15;
          f2_6= t5;
          f0_4 = f_ 15;
          f1_5 = f_26;
 LL2:     t3 = i_7;
          t4 = t_3 + 1;
          i_7 = t4;
          t2 = i_7 <= m_2;
          if (t2) goto LL3;
 LL1:     goto LL5;
 LL4:     return m_2;
 LL5:     return f_26;

}
```

**Fig. 4** *Three-address code IR for source file fib.c*

Temporary variable identifiers inserted by the frontend start with letter 't'. All local identifiers have a unique numerical suffix. This particular IR format is generated by the LANCE C frontend [7]



```
(1) f0_4 = 0;
(2) f1_5 = 1;
(3) t1 = m_2 <= 1;
(4) if (t1) goto LL4;

(22) LL4:
(23) return m_2;

(9) LL3:
(10) t5 = f0_4 + f1_5;
(11) f2_6 = t5;
(12) f0_4 = f1_5;
(13) f1_5 = f2_6;

(5) i_7 = 2;
(6) t2 = i_7 <= m_2;
(7) t6 = !t2;
(8) if (t6) goto LL1;

(14) LL2:
(15) t3 = i_7;
(16) t4 = t3 + 1;
(17) i_7 = t4;
(18) t2 = i_7 <= m_2;
(19) if (t2) goto LL3;

(20) LL1:
(21) goto LL5;

(24) LL5:
(25) return f2_6;
```

**Fig. 5** *Control flow graph for fib.c*

For instance, the data flow graph (DFG) from Fig. 6 shows relations of the form 'statement X computes a value used as an argument in statement Y'.

Both the CFG and the DFG form the basis for many code optimisation passes at the IR level. These include common subexpression elimination, jump optimisation, loop-invariant code motion, dead code elimination and other 'Dragon Book' [1] techniques. Owing to their target machine independence, these IR optimisations are generally considered complementary to machine code generation in the backend and are supposed to be useful 'on average' for any type of target. However, care must be taken to select an appropriate IR optimisation sequence or script for each particular target, since certain (sometimes quite subtle) machine dependencies do exist. For instance, common subexpression elimination removes redundant computations to save execution time and code size. At the assembly level, however, this effect might be over-compensated by the higher register pressure that increases the amount of spill code. Moreover, there are many interdependencies between the IR optimisations themselves. For instance, constant propagation generally creates new optimisation opportunities for constant folding, and vice versa, and dead code elimination is frequently required as a 'cleanup' phase between other IR optimisations. A poor choice of IR optimisations can have a dramatic effect on final code quality. Thus, it is important that IR optimisations be organised in a modular fashion, so as to permit enabling and disabling of particular passes during fine-tuning of a new compiler.

## 2.3 Machine code generation

During this final compilation phase, the IR is mapped to target assembly code. Since for a given IR an infinite number of mappings as well as numerous constraints exist, this is clearly a complex optimisation problem. In fact, even many optimisation subproblems in code generation are NP-hard, i.e. require exponential runtime for optimal solutions. As a divide-and-conquer approach, the backend is thus generally organised into three main phases: code selection, register allocation and scheduling, which are implemented with a variety of heuristic algorithms. Dependent on the exact problem definition, all of these phases may be considered NP-hard, e.g. [8] analyses the complexity of code generation for certain types of target machines.

For the purpose of code selection, the optimised IR is usually converted into a sequence of tree-shaped DFGs. Using a cost metric for machine instructions, the code selector aims at a minimum-cost covering of the DFGs by instruction patterns (Figs. 7 and 8). In particular for target architectures with complex instruction sets, such as CISCs and DSPs, careful code selection is the key to good code quality.

For complexity reasons, most code selectors work only on trees [9], even though generalised code selection for arbitrary DFGs can yield higher code quality for certain architectures [10, 11]. The computational effort for solving the NP-hard generalised code selection problem is normally considered too high in practice, though.

Subsequent to code selection, the register allocator decides which variables are to be kept in machine registers to ensure efficient access. Careful register allocation is key for target machines with RISC-like load-store architectures and large register files. Frequently, there are many more simultaneously live variables than physically available machine registers. In this case, the register allocator inserts

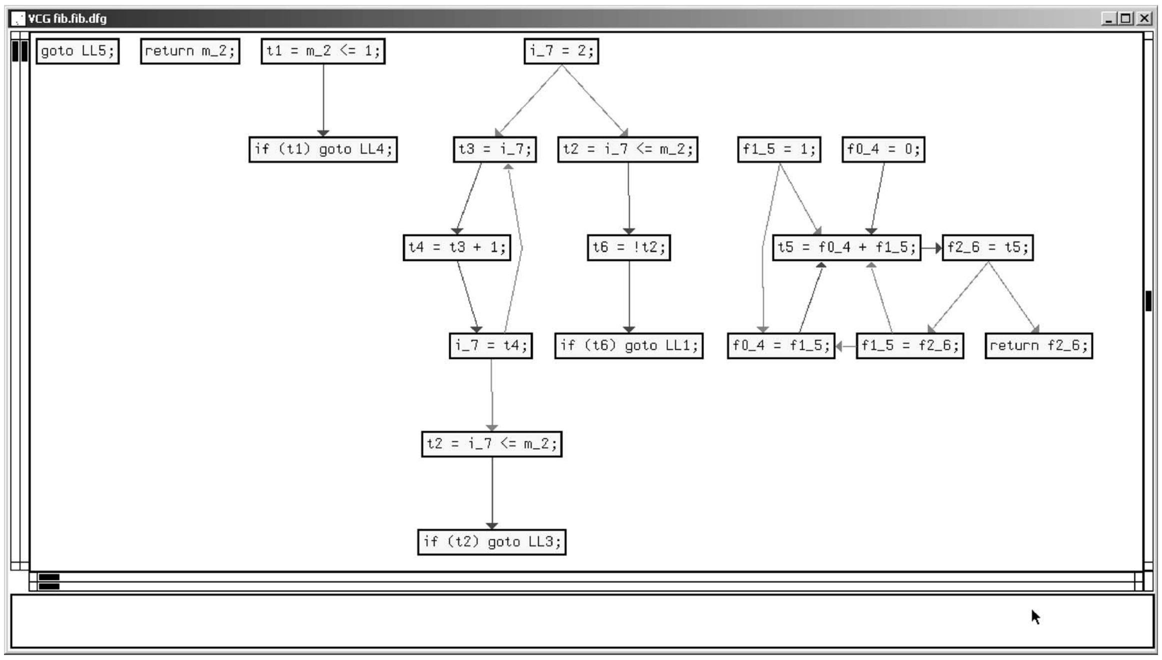


**Fig. 6** *Data flow graph for fib.c*

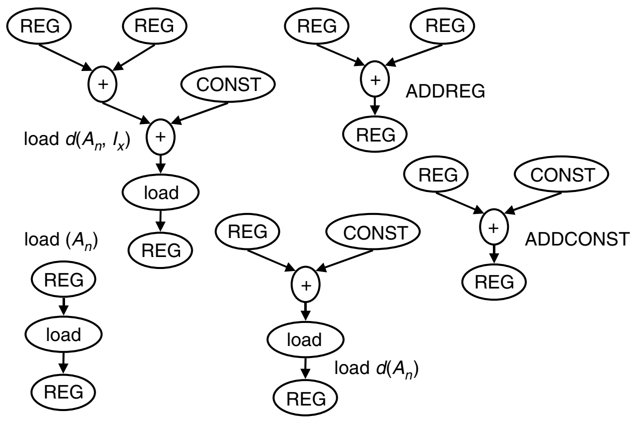


**Fig. 7** *Five instruction patterns available for a Motorola 68k CPU*

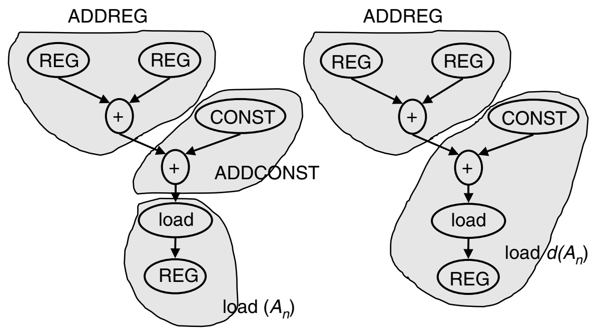


**Fig. 8** *Two possible coverings of a DFG using the instruction patterns from Fig. 7*

spill code to temporarily store register variables to main memory. Obviously, spill code needs to be minimised in order to optimise program performance and code size. Many register allocators use a graph colouring approach [12, 13] to accomplish this.

For target processors with instruction pipeline hazards and/or instruction-level parallelism, such as VLIW machines, instruction scheduling is required to optimise code execution speed. The scheduler requires two sets of inputs to represent the interinstruction constraints: instruction latencies and reservation tables. Both constraints refer to the schedulability of some pair (X, Y) of instructions. If instruction Y depends on instruction X, the latency values determine a lower bound on the number of cycles between execution of X and Y. If there is no dependence between X and Y, the occupation of exclusive resources represented by the reservation tables determines whether the execution of X may overlap the execution of Y in time, e.g. whether X and Y may be executed in parallel on a VLIW processor. As for other backend phases, optimal scheduling is an intractable problem. However, there exist a number of powerful scheduling heuristics, such as (local) list scheduling and (global) trace scheduling [3].

Besides the above three standard code generation phases, a backend frequently also incorporates different target-specific code optimisation passes. For instance, address code optimisation [14] is useful for a class of DSPs, so as to fully utilise dedicated address generation hardware for pointer arithmetic. Many VLIW compilers employ loop unrolling and software pipelining [15] for increasing instruction-level parallelism in the hot spots of application code. Loop unrolling generates larger basic blocks inside loop bodies, and hence provides better opportunities for keeping the VLIW functional units busy most of the time. Software pipelining rearranges the loop iterations so as to remove intraloop data dependencies that otherwise would obstruct instruction-level parallelism. Finally, NPU architectures for efficient protocol processing require yet a different set of machine-specific techniques [16] that exploit bit-level manipulation instructions.

The separation of the backend into multiple phases is frequently needed to achieve sufficient compilation speed, but tends to compromise code quality due to interdependencies between the phases. In particular, this holds for irregular 'non-RISC' instruction sets, where the phase interdependencies are sometimes very tight. Although there have been attempts to solve the code generation problem in its entirety, e.g. based on integer linear programming [17], such 'phase-coupled' code generation techniques are still not in widespread use in real-word compilers.

## 3 Approaches to retargetable compilation

From the above discussions is it obvious that compiler retargeting mainly requires adaptations of the backend, even though IR optimisation issues certainly should not be neglected. In order to provide a retargetable compiler with a processor model, as sketched in Fig. 1, a formal machine description language is required. For this purpose, dozens of different approaches exist. These can be classified with respect to the intended target processor class (e.g. RISC against VLIW) and the modelling abstraction level, e.g. purely behavioural, compiler-oriented against more structural, architecture-oriented modelling styles.

Behavioural modelling languages make the task of retargeting easier, because they explicitly capture compiler-related information about the target machine, i.e. instruction set, register architecture and scheduling constraints. However, they usually require good understanding of compiler technology. In contrast, architectural modelling languages follow a more hardware design oriented approach and describe the target machine in more detail. This is convenient for users not so familiar with compiler technology. However, automatic retargeting gets more difficult, because a ‘compiler view’ needs to be extracted from the architecture model, while eliminating unnecessary details.

In the following, we will briefly discuss a few representative examples of retargetable compiler systems. For a comprehensive overview of existing systems see [18].

### 3.1 MIMOLA

MIMOLA denotes both a mixed programming and hardware description language (HDL) and a hardware design system. As the MIMOLA HDL serves multiple purposes, e.g. register-transfer level (RTL) simulation and synthesis, the retargetable compiler MSSQ [19, 20] within the MIMOLA design system follows the above-mentioned architecture-oriented approach. The target processor is described as an RTL netlist, consisting of components and interconnect. Figure 9 gives an example of such an RTL model.

Since the HDL model comprises all RTL information about the target machine’s controller and data path, it is clear that all information relevant for the compiler backend of MSSQ is present too. However, this information is only implicitly available, and consequently the lookup of this information is more complicated than in a behavioural model.

```
MODULE SimpleProcessor (IN inp:(7:0); OUT outp:(7:0)); STRUCTURE
IS TYPE InstrFormat = FIELDS      -- 21-bit horizontal instruction word
           imm:       (20:13);
           RAMadr:     (12:5);
           RAMctr:        (4);
           mux:         (3:2);
           alu:         (1:0);
         END;
        Byte = (7:0); Bit = (0);  -- scalar types

PARTS                           -- instantiate behavioral modules
 IM: MODULE InstrROM (IN adr: Byte; OUT ins: InstrFormat);
      VAR storage: ARRAY[0..255] OF InstrFormat;
      BEGIN ins <- storage[adr]; END;
 PC, REG: MODULE Reg8bit (IN data: Byte; OUT outp: Byte);
           VAR R: Byte;
           BEGIN R := data; outp <- R; END;
 PCIncr: MODULE IncrementByte (IN data: Byte; OUT inc: Byte);
          BEGIN outp <- INCR data; END;
 RAM: MODULE Memory (IN data, adr: Byte; OUT outp: Byte; FCT c: Bit);
       VAR storage: ARRAY[0..255] OF Byte;
       BEGIN
        CASE c OF: 0: NOLOAD storage; 1: storage[adr] := data; END;
        outp <- storage[adr];
       END;
 ALU: MODULE AddSub (IN d0, d1: Byte; OUT outp: Byte; FCT c: (1:0));
       BEGIN                  -- "%" denotes binary numbers
        outp <- CASE c OF %00: d0 + d1; %01: d0 - d1; %1x: d0; END;
       END;
 MUX: MODULE Mux3x8 (IN d0,d1,d2: Byte; OUT outp: Byte; FCT c: (1:0));
       BEGIN outp <- CASE c OF 0: d0;  1: d1; ELSE: d2; END; END;

CONNECTIONS
 -- controller:                      -- data path:
 PC.outp       -> IM.adr;            IM.ins.imm    -> MUX.d0;
 PC.outp       -> PCIncr.data;       inp       -> MUX.d1;   -- primary input
 PCIncr.outp   -> PC.data;           RAM.outp  -> MUX.d2;
 IM.ins.RAMadr -> RAM.adr;           MUX.outp  -> ALU.d1;
 IM.ins.RAMctr -> RAM.c;             ALU.outp  -> REG.data;
 IM.ins.alu    -> ALU.c;             REG.outp  -> ALU.d0;
 IM.ins.mux    -> MUX.c;             REG.outp  -> outp;     -- primary output
END; -- STRUCTURE LOCATION_FOR_PROGRAMCOUNTER PC;
LOCATION_FOR_INSTRUCTIONS IM; END; -- STRUCTURE
```

**Fig. 9** *MIMOLA HDL model of a simple processor*

MSSQ compiles an ‘extended subset’ of the PASCAL programming language directly into binary machine code. Due to its early introduction, MSSQ employs only few advanced code optimisation techniques (e.g. there is no graph-based global register allocation), but performs the source-to-architecture mapping in a straightforward fashion, on a statement-by-statement basis. Each statement is represented by a data flow graph (DFG), for which an isomorphic subgraph is searched in the target data path. If this matching fails, the DFG is partitioned into simpler components, for which graph matching is invoked recursively.

In spite of this simple approach, MSSQ is capable of exploiting instruction-level parallelism in VLIW-like architectures very well, due to the use of a flexible instruction scheduler. However, code quality is generally not acceptable in the case of complex instruction sets and load/store data paths. In addition, it shows comparatively high compilation times, due to the need for exhaustive graph matching.

The MIMOLA approach shows very high flexibility in compiler retargeting since in principle any target processor can be represented as an RTL HDL model. In addition, it avoids the need to consistently maintain multiple different models of the same machine for different design phases, e.g. simulation and synthesis, as all phases can use the same ‘golden’ reference model. MSSQ demonstrates that retargetable compilation is possible with such unified models, even though it does not handle well architectures with complex instruction pipelining constraints (which is a limitation of the tool, though, rather than of the approach itself). The disadvantage, however, is that the comparatively detailed modelling level makes it more difficult to develop the model and to understand its interaction with the retargetable compiler, since e.g. the instruction set is ‘hidden’ in the model.

Some of the limitations have been removed in RECORD, another MIMOLA HDL based retargetable compiler that comprises dedicated code optimisations for DSPs. In order to optimise compilation speed, RECORD uses an instruction set extraction technique [21] that bridges the gap between RTL models and behavioural processor models. Key ideas of MSSQ, e.g. the representation of scheduling constraints by binary partial instructions, have also been adopted in the CHESS compiler [22, 23], one of the first commercial tool offerings in that area. In the Expression compiler [24], the concept of structural architecture modelling has been further refined to increase the reuse opportunities for model components.

## 3.2 GNU C compiler

The widespread GNU C compiler gcc [25] can be retargeted by means of a machine description file that captures the compiler view of a target processor in a behavioural fashion. In contrast to MIMOLA, this file format is heterogeneous and solely designed for compiler retargeting. The gcc compiler is organised into a fixed number of different passes. The frontend generates a three-address code like intermediate representation (IR). There are multiple built-in ‘Dragon Book’ IR optimisation passes, and the backend is driven by a specification of instruction patterns, register classes and scheduler tables. In addition, retargeting gcc requires C code specification of numerous support functions, macros and parameters.

The gcc compiler is robust and well-supported, it includes multiple source language frontends and it has been ported to dozens of different target machines, including typical embedded processor architectures like ARM, ARC, MIPS and Xtensa. However, it is very complex and hard to customise. It is primarily designed for ‘compiler-friendly’ 32-bit RISC-like load-store architectures. While porting to more irregular architectures, such as DSPs, is possible as well, this generally results in huge retargeting effort and/or insufficient code quality.

## 3.3 Little C compiler

Like gcc, retargeting the ‘little C compiler’ lcc [26, 27] is enabled via a machine description file. In contrast to gcc, lcc is a ‘lightweight’ compiler that comes with much less source code and only a few built-in optimisations, and hence lcc can be used to design compilers for certain architectures very quickly. The preferred range of target processors is similar to that of gcc, though with some further restrictions on irregular architectures.

In order to retarget lcc, the designer has to specify the available machine registers, as well as the translation of C operations (or IR operations, respectively) to machine instructions by means of mapping rules. The following excerpt from lcc’s Sparc machine description file [27] exemplifies two typical mapping rules:

```
addr: ADDP4 (reg, reg) "%%%0 + %%%1"
reg: INDIRI1 (addr) "ldsb [%0], %%%c\n"
```

The first line instructs the code selector how to cover address computations (‘addr’) that consist of adding two 4-byte pointers (‘ADDP4’) stored in registers (‘reg’). The string ‘%%%0 + %%%1’ denotes the assembly code to be emitted, where ‘%0’ and ‘%1’ serve as placeholders for the register numbers to be filled later by the register allocator (and ‘%%’ simply emits the register identifier symbol ‘%’). Since ‘addr’ is only used in context with memory accesses, here only a substring without assembly mnemonics is generated.

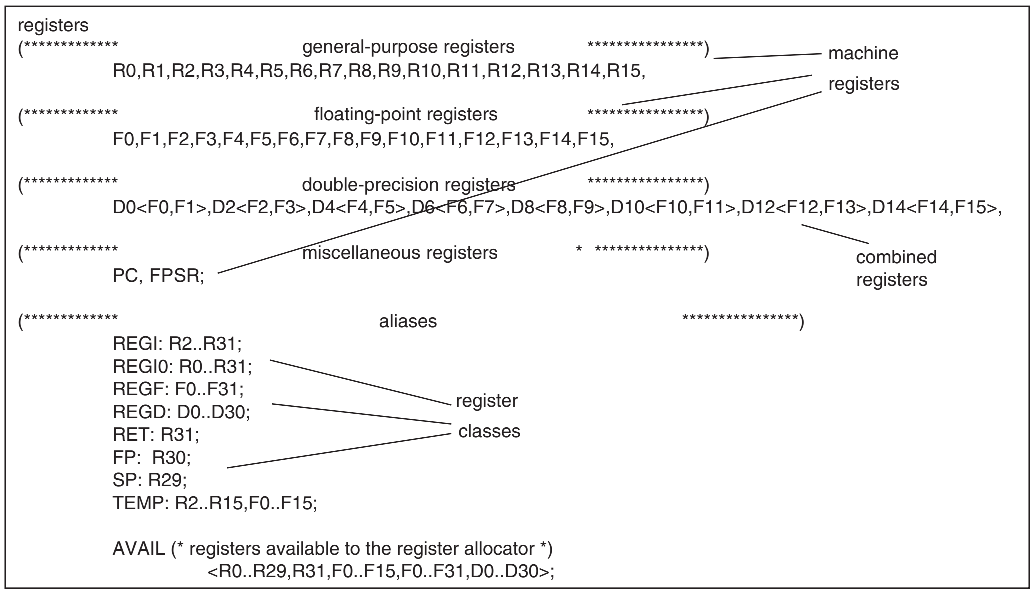
The second line shows the covering of a 1-byte signed integer load from memory (‘INDIRI1’), which can be implemented by assembly mnemonic ‘ldsb’, followed by arguments referring to the load address (‘%0’, returned from the ‘addr’ mapping rule) and the destination register (‘%c’).

By specifying such mapping rules for all C/IR operations plus around 20 relatively short C support functions, lcc can be retargeted quite efficiently. However, lcc is very limited in the context of non-RISC embedded processor architectures. For instance, it is impossible to model certain irregular register architectures (as e.g. in DSPs) and there is no instruction scheduler, which is a major limitation for targets with instruction level parallelism. Therefore, lcc has not found wide use in code generation for embedded processors so far.

## 3.4 CoSy

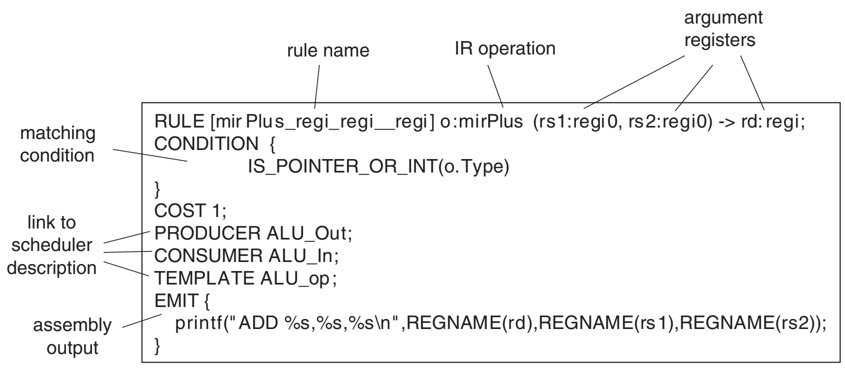
The CoSy system from ACE [28] is a retargetable compiler for multiple source languages, including C and C++. Like gcc, it includes several Dragon Book optimisations, but shows a more modular, extensible software architecture, which permits the addition of IR optimisation passes through well-defined interfaces.

For retargeting, CoSy comprises a backend generator that is driven by the *CGD machine description format*. Similar to gcc and lcc, this format is full-custom and only designed for use in compilation. Hence, retargeting CoSy requires significant compiler know-how, particularly with respect to code selection and scheduling. Although it generates the backend automatically from the CGD specification, including standard algorithms for code selection, register



```
registers
(*************          general-purpose registers          ****************)
          R0,R1,R2,R3,R4,R5,R6,R7,R8,R9,R10,R11,R12,R13,R14,R15,

(*************          floating-point registers           ****************)
          F0,F1,F2,F3,F4,F5,F6,F7,F8,F9,F10,F11,F12,F13,F14,F15,

(*************          double-precision registers         ****************)
          D0<F0,F1>,D2<F2,F3>,D4<F4,F5>,D6<F6,F7>,D8<F8,F9>,D10<F10,F11>,D12<F12,F13>,D14<F14,F15>,

(*************          miscellaneous registers          * ***************)
          PC, FPSR;

(*************                    aliases                          ****************)
         REGI: R2..R31;
         REGI0: R0..R31;
         REGF: F0..F31;
         REGD: D0..D30;
         RET: R31;
         FP:  R30;
         SP: R29;
         TEMP: R2..R15,F0..F15;

         AVAIL (* registers available to the register allocator *)
                  <R0..R29,R31,F0..F15,F0..F31,D0..D30>;
```

**Fig. 10** *CGD specification of processor registers*



```
RULE [mir Plus_regi_regi__regi] o:mirPlus  (rs1:regi0, rs2:regi0) -> rd: regi;
CONDITION  {
        IS_POINTER_OR_INT(o.Type)
}
COST 1;
PRODUCER ALU_Out;
CONSUMER ALU_In;
TEMPLATE ALU_op;
EMIT {
  printf("ADD %s,%s,%s\n",REGNAME(rd),REGNAME(rs1),REGNAME(rs2));
}
```

rule name
IR operation
argument
registers
matching
condition
link to
scheduler
description
assembly
output

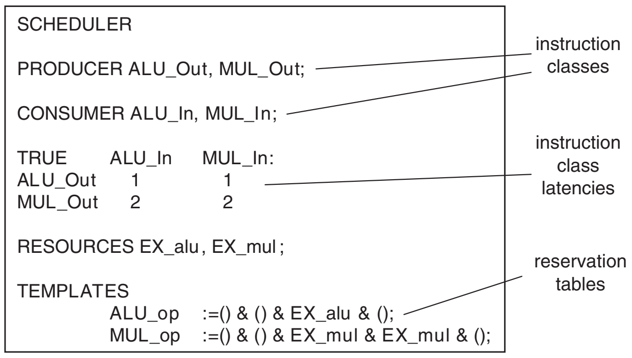
**Fig. 11** *CGD specification of mapping rules*

allocation and scheduling, the designer has to fully understand the IR-to-assembly mapping and how the architecture constrains the instruction scheduler.

The CGD format follows the classical backend organisation. It includes mapping rules, a register specification, as well as scheduler tables. The register specification is a straightforward listing of the different register classes and their availability for the register allocator (Fig. 10).

Mapping rules are the key element of CGD (Fig. 11). Each rule describes the assembly code to be emitted for a certain C/IR operation, depending on matching conditions and cost metric attributes. Similar to gcc and lcc, the register allocator later replaces symbolic registers with physical registers in the generated code.

Mapping rules also contain a link to the CGD scheduler description. By means of the keywords 'PRODUCER' and 'CONSUMER', the instructions can be classified into groups, so as to make the scheduler description more compact. For instance, arithmetic instructions performed on a certain ALU generally have the same latency values. In the scheduler description itself (Fig. 12), the latencies for pairs of instruction groups are listed as a table of numerical values. As explained in Section 2.3, these values instruct the scheduler to arrange instructions a minimum amount of cycles apart from each other. Different types of interinstruction dependencies are permitted. Here the keyword 'TRUE' denotes data dependence. (Data dependencies are sometimes called 'true', since they are induced by the source program itself. Hence, they cannot be removed by the



```
SCHEDULER

PRODUCER ALU_Out, MUL_Out;

CONSUMER ALU_In, MUL_In;

TRUE        ALU_In      MUL_In:
ALU_Out      1            1
MUL_Out      2            2

RESOURCES EX_alu, EX_mul;

TEMPLATES
            ALU_op     :=() & () & EX_alu & ();
            MUL_op     :=() & () & EX_mul & EX_mul & ();
```

instruction
classes
instruction
class
latencies
reservation
tables

**Fig. 12** *CGD specification of scheduler tables*

compiler. In contrast, there are 'false', or antidependencies that are only introduced by code generation via reuse of registers for different variables. The compiler should aim at minimising the amount of false dependencies, in order to maximise the instruction scheduling freedom).

Via the 'TEMPLATE' keyword, a reservation table entry is referenced. The '&' symbol separates the resource use of an instruction group over the different cycles during its processing in the pipeline. For instance, in the last line of Fig. 12, instruction group 'MUL_op' occupies resource 'EX_mul' for two subsequent cycles.

The CGD processor modelling formalism makes CoSy a quite versatile retargetable compiler. Case studies for RISC architectures show that the code quality produced by CoSy compilers is comparable to that of gcc. However, the complexity of the CoSy system and the need for compiler background knowledge make retargeting more tedious than, for example, in the case of lcc.

## 4 Processor architecture exploration

### 4.1 Methodology and tools for ASIP design

As pointed out in Section 1, one of the major applications of retargetable compilers in SoC design is to support the design and programming of application-specific instruction set processors (ASIPs). ASIPs receive increasing attention in both academia and industry due to their optimal flexibility/efficiency compromise [29]. The process of evaluating and refining an initial architecture model step-by-step to optimise the architecture for a given application is commonly called *architecture exploration*. Given that the ASIP application software is written in a high-level language like C, it is obvious that compilers play a major role in architecture exploration. Moreover, in order to permit frequent changes of the architecture during the exploration phase, compilers have to be retargetable.

Today's most widespread architecture exploration methodology is sketched in Fig. 13. It is an iterative approach that requires multiple remapping of the application code to the target architecture. In each iteration, the usual software development tool chain (C compiler, assembler, linker) is used for this mapping. Since exploration is performed with a virtual prototype of the architecture, an instruction set simulator together with a profiler are used to measure the efficiency and cost of the current architecture with respect to the given (range of) applications, e.g. in terms of performance and area requirements.

We say that hardware (processor architecture and instruction set) and software (application code) 'match', if the hardware meets the performance and cost goals, and there is no over- or underutilisation of HW resources. For instance, if the HW is not capable of executing the 'hot spots' of the application code under the given timing constraints, e.g. due to insufficient function units, too much spill code, or too many pipeline stalls, then more resources need to be provided. However, if many function units are idle most of the time or half of the register file remains unused, this indicates an underutilisation. Fine-grained profiling tools make such data available to the processor designer. However, it is still a highly creative process to determine the exact source of bottlenecks (application code, C compiler, processor instruction set, or microarchitecture)

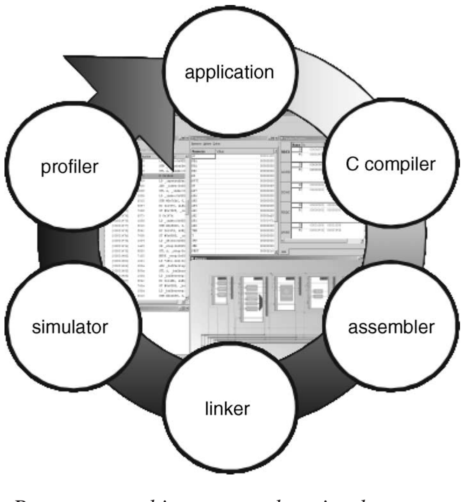


**Fig. 13** *Processor architecture exploration loop*

and to remove them by corresponding modifications, while simultaneously overlooking their potential side effects.

If the HW/SW match is initially not satisfactory, the ASIP architecture is further optimised, dependent on the bottlenecks detected during simulation and profiling. This optimisation naturally requires hardware design knowledge, and may comprise, for example, addition of application-specific custom machine instructions, varying register file sizes, modifying the pipeline architecture, adding more function units to the data path, or simply removing unused instructions. The exact consequences of such modifications are hard to predict, so that usually multiple iterations are required in order to arrive at an optimal ASIP architecture that can be handed over to synthesis and fabrication.

With the research foundations of this methodology laid in the 1980s and 1990s (see [18] for a summary of early tools), several commercial offerings are available now in the EDA industry, and more and more start-up companies are entering the market in that area. While ASIPs offer many advantages over off-the-shelf processor cores (e.g. higher efficiency, reduced royalty payments and better product differentiation), a major obstacle is still the potentially costly design and verification process, particularly concerning the software tools shown in Fig. 13. In order to minimise these costs and to make the exploration loop efficient, all approaches to processor architecture exploration aim at automating the retargeting of these tools as much as possible. In addition, a link to hardware design has to be available in order to accurately estimate area, cycle time and power consumption of a new ASIP. In most cases this is enabled by automatic HDL generation capabilities for processor models, which provides a direct entry to gate-true estimations via traditional synthesis flows and tools.

One of the most prominent examples of an industrial ASIP is the Tensilica Xtensa processor [30]. It provides a basic RISC core that can be extended and customised by adding new machine instructions and adjusting parameters, e.g. for the memory and register file sizes. Software development tools and an HDL synthesis model can be automatically generated. Application programming is supported via the gcc compiler and a more optimising in-house C compiler variant. The Tensilica Xtensa, together with its design environment, completely implement the exploration methodology from Fig. 13. However, the use of a largely predefined RISC core as the basic component poses limitations on the flexibility and the permissible design space. An important new entry to the ASIP market is Stretch [31]. Their configurable S5000 processor is based on the Xtensa core, but includes an embedded field programmable gate array (FPGA) for processor customisation. While FPGA vendors have combined processors and configurable logic on a single chip for some time, the S5000 'instruction set extension fabric' is optimised for implementation of custom instructions, thus providing a closer coupling between processor and FPGA. In this way, the ASIP becomes purely software-configurable and field-programmable, which reduces the design effort, but at the expense of reduced flexibility.

### 4.2 ADL-based approach

More flexibility is offered by the tool suite from Target Compiler Technologies [23] that focuses on the design of ASIPs for signal processing applications. In this approach, the target processor can be freely defined by the user in the nML architecture description language (ADL). In contrast to a purely compiler-specific machine model, such as in the case of gcc or CoSy's CGD, an ADL such as nML also captures information relevant for the generation of other software development tools, e.g. simulator, assembler and debugger, and hence covers a greater level of detail. However, in contrast to HDL-based approaches to retargetable compilation, such as MIMOLA, the abstraction level is still higher than RTL and usually allows for a concise explicit modelling of the instruction set. The transition to RTL only takes place once the ADL model is refined to an HDL model for synthesis.

LISATek is another ASIP design tool suite that originated at Aachen University [32]. It has first been produced by LISATek Inc. and is now available as a part of CoWare's SoC design tool suite [33]. LISATek uses the LISA 2.0 (language for instruction set architectures) ADL for processor modelling. A LISA model captures the processor resources like registers, memories and instruction pipelines, as well as the machine's instruction set architecture (ISA). The ISA model is composed of operations (Fig. 14), consisting of sections that describe the binary coding, timing, assembly syntax and behaviour of machine operations at different abstraction levels. In an instruction-accurate model (typically used for early architecture exploration), no pipeline information is present, and each operation corresponds to one instruction. In a more fine-grained, cycle-accurate model, each operation represents a single pipeline stage of one instruction. LISATek permits the generation of software development tools (compiler, simulator, assembler, linker, debugger, etc.) from a LISA model, and embeds all tools into an integrated GUI environment for application and architecture profiling. In addition, it supports the translation of LISA models to synthesisable VHDL and Verilog RTL models. Figure 15 shows the intended ASIP design flow with LISATek. In addition to an implementation of the exploration loop from Fig. 13, the flow also comprises the synthesis path via HDL models, which enables back-annotation of gate-level hardware metrics.

```
OPERATION ADD IN pipe.EX {
  // declarations
  DECLARE {
   INSTANCE writeback;
   GROUP src1, dst = { reg };
   GROUP src2 = { reg || imm };}

  // assembly syntax
  SYNTAX { "addc" dst "," src1 "," src2 }

  // binary encoding
  CODING { 0b0101 dst src1 src2 }

  // behavior (C code)
  BEHAVIOR {
   u32 op1, op2, result, carry;
   if (forward) {
    op1 = PIPELINE_REGISTER(pipe,EX/WB).result;}
   else {
    op1 = PIPELINE_REGISTER(pipe,DC/EX).op1;}
   result = op1 + op2;
   carry = compute_carry(op1, op2, result);
   PIPELINE_REGISTER(EX/WB).result = result;
   PIPELINE_REGISTER(EX/WB).carry = carry; }

  // pipeline timing
  ACTIVATION { writeback, carry_update }
}
```

**Fig. 14** *LISA operation example: execute stage of an ADD instruction in a cycle-true model with forwarding hardware modelling*

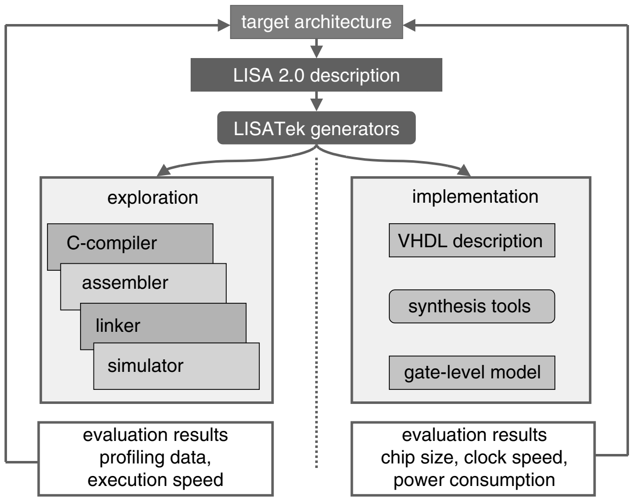


**Fig. 15** *LISATek-based ASIP design flow*

In [34] it has been exemplified how the LISATek architecture exploration methodology can be used to optimise the performance of an ASIP for an IPv6 security application. In this case study, the goal was to enhance a given processor architecture (MIPS32) by means of dedicated machine instructions and a microarchitecture for fast execution of the compute-intensive Blowfish encryption algorithm (Fig. 16) in IPsec. Based on initial application C code profiling, hot spots were identified that provided first hints on appropriate custom instructions. The custom instructions were implemented as a coprocessor (Fig. 17) that communicates with the MIPS main processor via shared memory. The coprocessor instructions were accessed from the C compiler generated from the LISA model via compiler intrinsics. This approach was feasible due to the small

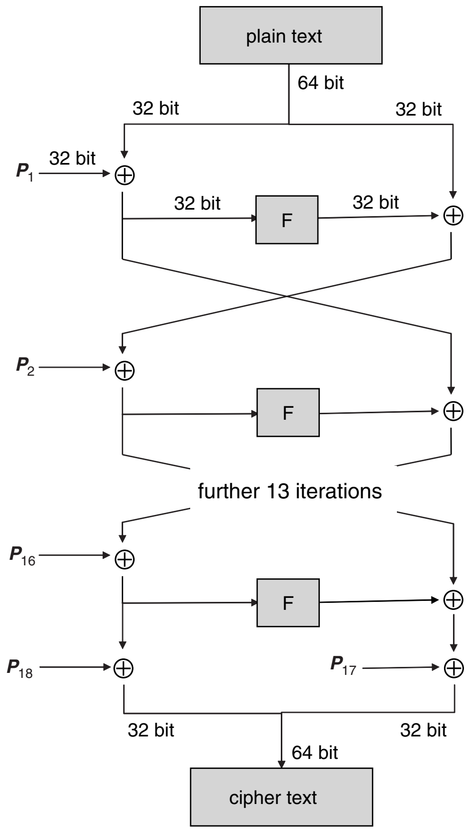


**Fig. 16** *Blowfish encryption algorithm for IPsec*

$P_i$ denotes a 32-bit subkey, F denotes the core subroutine consisting of substitutions and add/xor operations

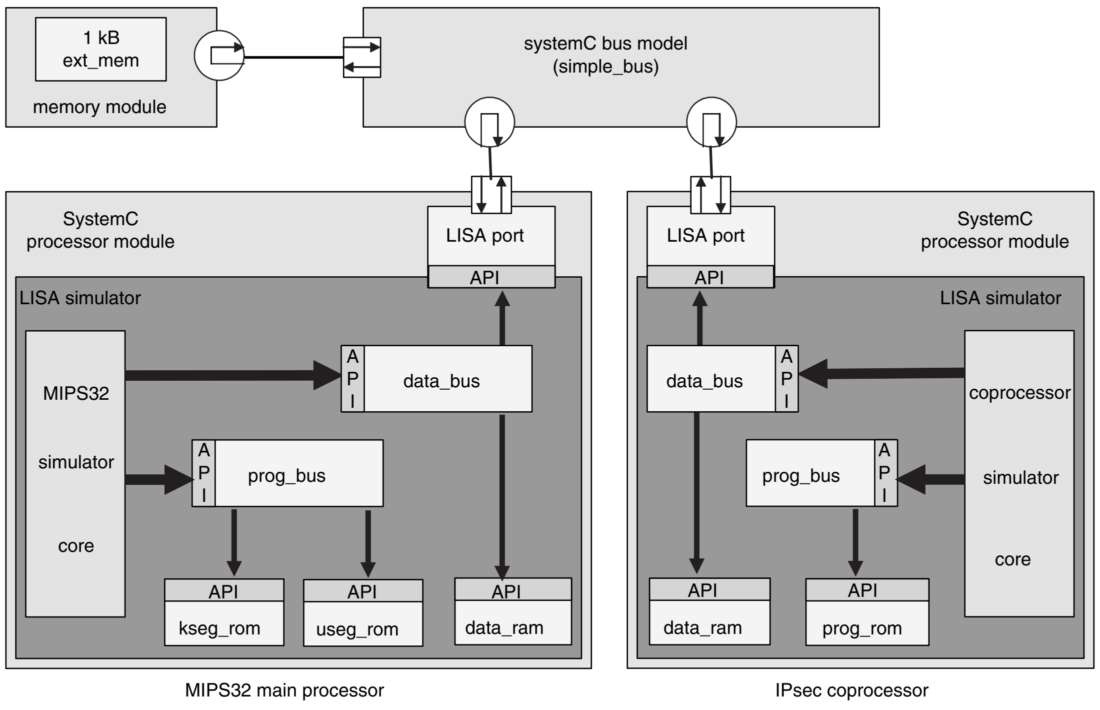


**Fig. 17** *MIPS32/coprocessor system resulting from Blowfish architecture exploration*

number of custom instructions required, which can be easily utilised with small modifications of the initial Blowfish C source code. LISATek-generated instruction-set simulators embedded into a SystemC based cosimulation environment were used to evaluate candidate instructions and to optimise the coprocessor's pipeline microarchitecture on a cycle-accurate level.

Finally, the architecture implementation path via LISA-to-VHDL model translation and gate-level synthesis was used for further architecture fine-tuning. The net result was a 5× speedup of Blowfish execution over the original MIPS at the expense of an additional coprocessor area of 22 k gates. This case study demonstrates that ASIPs can provide excellent efficiency combined with IP reuse opportunities for similar applications from the same domain. Simultaneously, the iterative, profiling-based exploration methodology permits us to achieve such results quickly, i.e. typically within a few man-weeks.

The capability of modelling the ISA behaviour in arbitrary C/C++ code makes LISA very flexible with respect to different target architectures and enables the generation of high-speed ISA simulators based on the JITCC technology [35]. As in the MIMOLA and Target approaches, LISATek follows the 'single golden model' paradigm, i.e. only one ADL model (or automatically generated variants of it) is used throughout the design flow in order to avoid consistency problems and to guarantee 'correct-by-construction' software tools during architecture exploration. Under this paradigm, the construction of retargetable compilers is a challenging problem, since in contrast to special-purpose languages like CGD, the ADL model is not tailored towards compiler support only. Instead, similar to MIMOLA/MSSQ (see Section 3.1), the compiler-relevant information needs to be extracted with special techniques. This is discussed in more detail in the following Section.

## 5 C compiler retargeting in the LISATek platform

### 5.1 Concept

The design goals for the retargetable C compiler within the LISATek environment were to achieve high flexibility and good code quality at the same time. Normally, these goals are contradictory, since the more the compiler can exploit knowledge of the range of target machines, the better is the code quality, and vice versa. In fact, this inherent trade-off has been a major obstacle for the successful introduction of retargetable compilers for quite some time.

However, a closer look reveals that this only holds for 'push-button' approaches to retargetable compilers, where the compiler is expected to be retargeted fully automatically once the ADL model is available. If compiler retargeting follows a more pragmatic user-guided approach (naturally at the cost of a slightly longer design time), then one can escape from the above dilemma. In the case of the LISA ADL, an additional constraint is the unrestricted use of C/C++ for operation behaviour descriptions. Due to the need for flexibility and high simulation speed, it is impossible to sacrifice this description vehicle. However, this makes it very difficult to automatically derive the compiler semantics of operations, due to large syntactic variances in operation descriptions. In addition, hardware-oriented languages like ADLs do not at all contain certain types of compiler-related information, such as C type bit widths, function calling conventions etc., which makes an interactive GUI-based retargeting environment useful.

In order to maximise the reuse of existing, well tried compiler technology and to achieve robustness for real-life applications, the LISATek C compiler builds on the CoSy system (Section 3.4) as a backbone. Since CoSy is capable of generating the major backend components (code selector, register allocator, scheduler) automatically, it is sufficient to generate the corresponding CGD fragments (see Figs. 10–12) from a LISA model in order to implement an entire retargetable compiler tool chain.

### 5.2 Register allocator and scheduler

Out of the three backend components, the register allocator, is the easiest one to retarget since the register information is explicit in the ADL model. As shown in Fig. 10, essentially only a list of register names is required, which can be largely copied from the resource declaration in the LISA model. Special cases (e.g. combined registers, aliases,

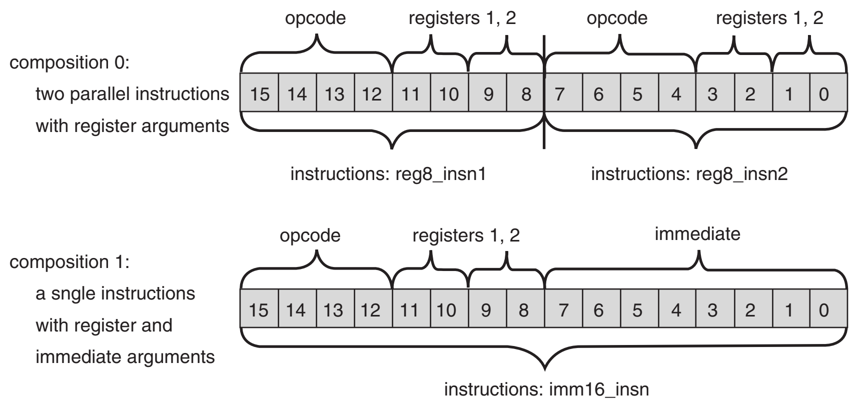


**Fig. 18** *Two instruction encoding formats (compositions)*

special-purpose registers such as the stack pointer) can be covered by a few one-time user interactions in the GUI.

As explained in Section 2.3, generation of the instruction scheduler is driven by two types of tables: latency tables and reservation tables. Both are only implicit in the ADL model. Reservation tables model interinstruction conflicts. Similar to the MSSQ compiler (Section 3.1), it is assumed that all such conflicts are represented by instruction encoding conflicts. (This means that parallel scheduling of instructions with conflicting resource usage is already prohibited by the instruction encoding itself. Architectures for which this assumption is not valid appear to be rare in practice, and if necessary there are still simple workarounds via user interaction, e.g. through manual addition of artificial resources to the generated reservation tables). Therefore, reservation tables can be generated by examining the instruction encoding formats in a LISA model.

Figures 18 and 19 exemplify the approach for two possible instruction formats or compositions. Composition 0 is VLIW-like and allows two parallel 8-bit instruction to be encoded. In composition 1, the entire 16 instruction bits are required due to an 8-bit immediate constant that needs to be encoded. In the corresponding LISA model, these two formats are modelled by means of a switch/case language construct.

The consequences of this instruction format for the scheduler are that instructions that fit into one of the 8-bit

```
OPERATION decode_op
{
 DECLARE
 {
   ENUM  composition = {composition0,
                        composition1};
   GROUP  reg8_insn1, reg8_insn2 = { reg8_op };
   GROUP imm16_insn = { imm16_op};
 }
 SWITCH(compositions)
 {
   CASE composition0:
   {
    CODING AT (progam_counter)
       {  insn_reg = = reg8_insn1 | | reg8_insn2 }
    SYNTAX { reg8_insn1 " , " reg_insn2}
   }
   CASE composition1:
   {
    CODING AT (progam_counter)
       {  insn_reg = = imm16_insn }
    SYNTAX { imm16_insn }
   }
 }
}
```

**Fig. 19** *LISA model fragment for instruction format from Fig. 18*

slots of composition 0 can be scheduled in either of the two, while an instruction that requires an immediate operand blocks other instructions from being scheduled in parallel. The scheduler generator analyses these constraints and constructs *virtual resources* to represent the interinstruction conflicts. Naturally, this concept can be generalised to handle more complex, realistic cases for wider VLIW instruction formats. Finally, a reservation table in CGD format (see Fig. 12) is emitted for further processing with CoSy. (We also generate a custom scheduler as an optional bypass of the CoSy scheduler. The custom one achieves better scheduling results for certain architectures [36].)

The second class of scheduler tables, latency tables, depends on the resource access of instructions as they run through the different instruction pipeline stages. In LISA, cycle-accurate instruction timing is described via activation sections inside the LISA operations. One operation can invoke the simulation of other operations downstream in the pipeline during subsequent cycles, e.g. an instruction fetch stage would typically be followed by a decode stage, and so forth. This explicit modelling of pipeline stages makes it possible to analyse the reads and writes to registers at a cycle-true level. In turn, this information permits the extraction of different types of latencies, e.g. due to a data dependency.

An example is shown in Fig. 20, where there is a typical four-stage pipeline (fetch, decode, execute, writeback) for a load/store architecture with a central register file. (The first stage in any LISA model is the ‘main’ operation that is called for every new simulation cycle, similar to the built-in semantics of the ‘main’ function in ANSI C). By tracing the operation activation chain, one can see that a given instruction makes two read accesses in stage ‘decode’ and a write access in stage ‘writeback’. For arithmetic instructions executed on the ALU, for instance, this implies a latency value of two (cycles) in the case of a data dependency. This information can again be translated into CGD scheduler latency tables (Fig. 12). The current version of the scheduler generator, however, is not capable of automatically analysing forwarding/bypassing hardware, which is frequently used to minimise latencies due to pipelining. Hence, the fine-tuning of latency tables is performed via user interaction in the compiler retargeting GUI, which allows the user to add specific knowledge in order to over-ride potentially too conservative scheduling constraints, so as to improve code quality.

## 5.3 Code selector

As sketched in Section 2.3, retargeting the code selector requires specification of instruction patterns (or mapping rules) used to cover a data flow graph representation of

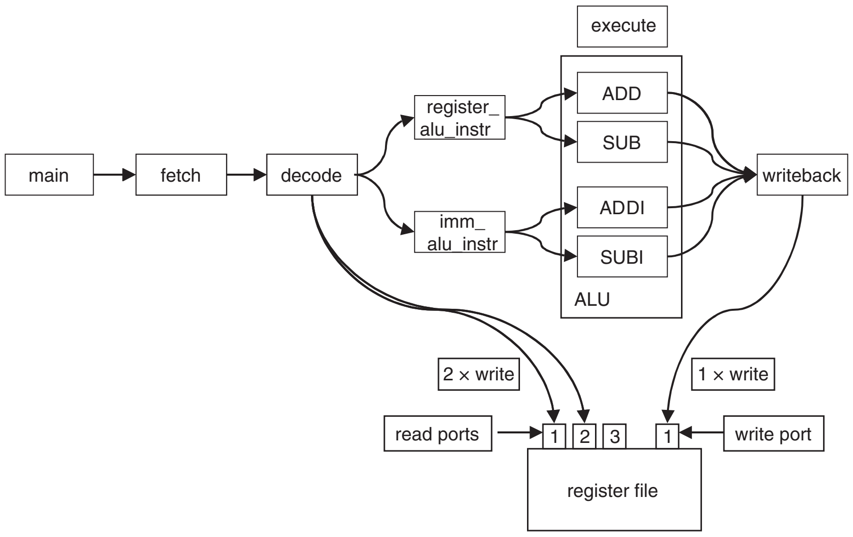
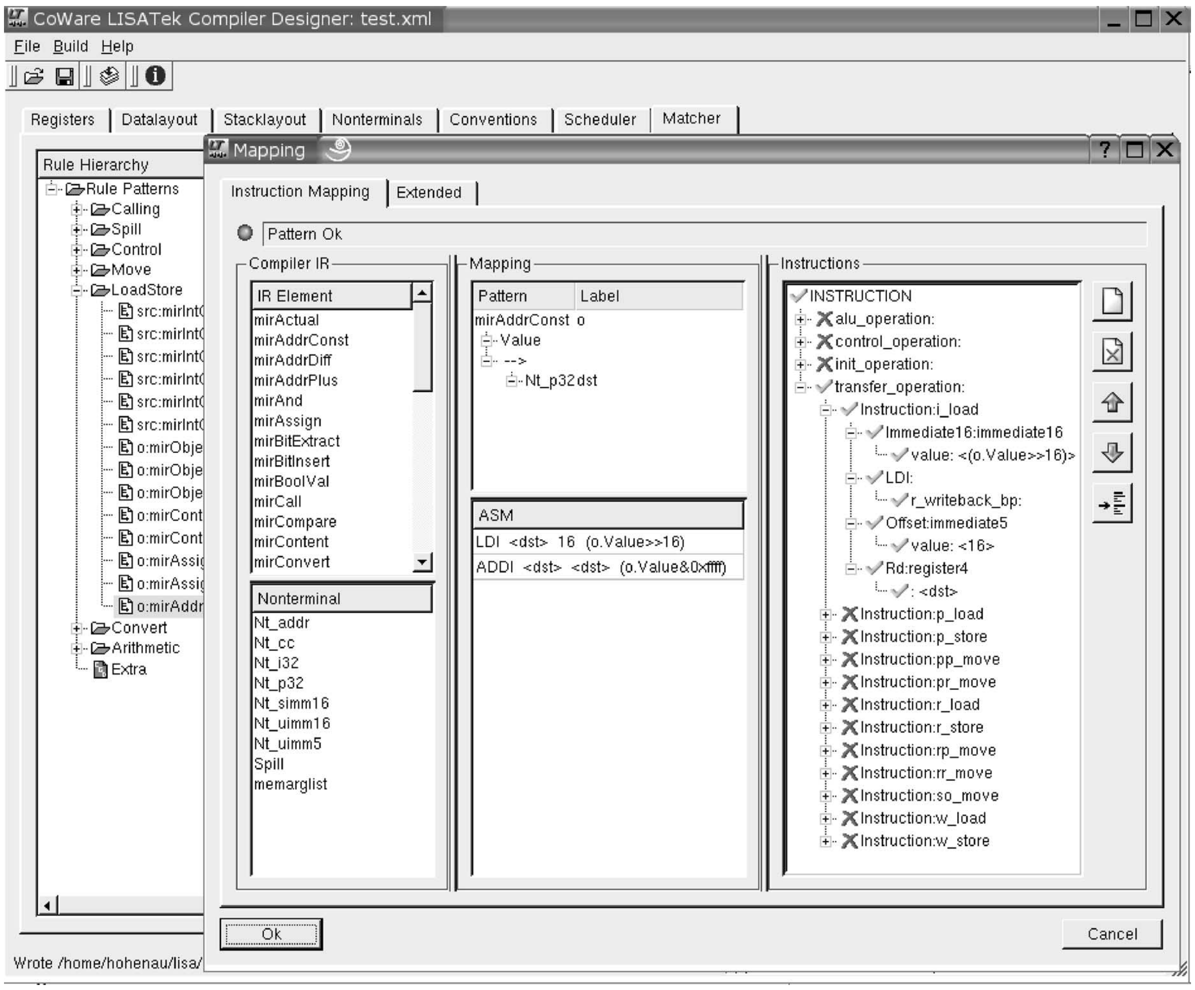


**Fig. 20** *Register file accesses of an instruction during its processing over different pipeline stages*

the intermediate representation (IR). Since it is difficult to extract instruction semantics from an arbitrary C/C++ specification as in the LISA behaviour models, this part of backend retargeting is least automated. Instead, the GUI offers the designer a mapping dialogue (Fig. 21, see also [37]) that allows for a manual specification of mapping rules. This dialogue enables the 'drag-and-drop' composition of mapping rules, based on (*a*) the IR operations needed to be covered for a minimal operational compiler and (*b*) the available LISA operations. In the example from Fig. 21, an address computation at the IR level is implemented with two target-specific instructions (LDI and ADDI) at the assembly level.

Although significant manual retargeting effort is required with this approach, it is much more comfortable than working with a plain compiler generator such as CoSy (Section 3.4), since the GUI hides many compiler internals from the user and takes the underlying LISA processor model explicitly into account, e.g. concerning the correct assembly syntax of instructions. Moreover, it ensures very



Rule Hierarchy
- Rule Patterns
  - Calling
  - Spill
  - Control
  - Move
  - LoadStore
    - src:mirInt(
    - src:mirInt(
    - src:mirInt(
    - src:mirInt(
    - src:mirInt(
    - src:mirInt(
    - o:mirObje
    - o:mirObje
    - o:mirObje
    - o:mirCont
    - o:mirCont
    - o:mirAssi
    - o:mirAssi
    - o:mirAddr
  - Convert
  - Arithmetic
  - Extra

Wrote /home/hohenau/lisa/

Pattern Ok

Compiler IR

| IR Element |
|---|
| mirActual |
| mirAddrConst |
| mirAddrDiff |
| mirAddrPlus |
| mirAnd |
| mirAssign |
| mirBitExtract |
| mirBitInsert |
| mirBoolVal |
| mirCall |
| mirCompare |
| mirContent |
| mirConvert |

| Nonterminal |
|---|
| Nt_addr |
| Nt_cc |
| Nt_i32 |
| Nt_p32 |
| Nt_simm16 |
| Nt_uimm16 |
| Nt_uimm5 |
| Spill |
| memarglist |

Mapping

| Pattern | Label |
|---|---|
| mirAddrConst o | |
| Value | |
| --> | |
| Nt_p32 dst | |

| ASM |
|---|
| LDI <dst> 16 (o.Value>>16) |
| ADDI <dst> <dst> (o.Value&0xffff) |

Instructions
- ✓ INSTRUCTION
  - ✗ alu_operation:
  - ✗ control_operation:
  - ✗ init_operation:
  - ✓ transfer_operation:
    - ✓ Instruction:i_load
      - ✓ Immediate16:immediate16
        - ✓ value: <(o.Value>>16)>
      - ✓ LDI:
        - ✓ r_writeback_bp:
      - ✓ Offset:immediate5
        - ✓ value: <16>
      - ✓ Rd:register4
        - ✓ : <dst>
    - ✗ Instruction:p_load
    - ✗ Instruction:p_store
    - ✗ Instruction:pp_move
    - ✗ Instruction:pr_move
    - ✗ Instruction:r_load
    - ✗ Instruction:r_store
    - ✗ Instruction:rp_move
    - ✗ Instruction:rr_move
    - ✗ Instruction:so_move
    - ✗ Instruction:w_load
    - ✗ Instruction:w_store

**Fig. 21** *GUI for interactive compiler retargeting (code selector view)*

high flexibility with respect to different target processor classes, and the user gets immediate feedback on consistency and completeness of the mapping specification.

The major drawback, however, of the above approach is a potential model consistency problem, since the LISA model is essentially overlayed with a (partially independent) code selector specification. In order to eliminate this problem, yet retaining flexibility, the LISA language has recently been enhanced with semantic sections [38]. These describe the behaviour of operations from a pure compiler perspective and in a canonical fashion. In this way, semantic sections eliminate syntactic variances and abstract from details such as internal pipeline register accesses or certain side effects that are only important for synthesis and simulation.

Figure 22 shows a LISA code fragment for an ADD instruction that generates a carry flag. The core operation ('dst = srcl + src2') could be analysed easily in this example, but in reality more C code lines might be required to capture this behaviour precisely in a pipelined model. The carry flag computation is modelled with a separate if-statement, but the detailed modelling style might obviously vary. However, the compiler only needs to know that the operation adds two registers and that it generates a carry, independent of the concrete implementation.

The corresponding semantics model (Fig. 23) makes this information explicit. Semantics models rely on a small set of precisely defined micro-operations ('_ADDI' for 'integer add' in this example) and capture compiler-relevant side effects with special attributes (e.g. '_C' for carry generation). This is feasible, since the meaning of generating

```
OPERATION ADD {
 DECLARE {
  GROUP src1, dst = { reg };
  GROUP src2 = { reg || imm };}
 SYNTAX { "add" dst "," src1 "," src2 }
 CODING { 0b0000 src1 src2 dst }
 BEHAVIOR {
  dst = src1 + src2;
  if (((src1 < 0) && (src2 < 0)) ||
      ((src1 > 0) && (src2 > 0) &&
       (dst < 0)) ||
      ((src1 > 0) && (src2 < 0) &&
       (src1 > -src2)) ||
      ((src1 < 0) && (src2 > 0) &&
       (-src1 < src2)))
  { carry = 1; }}}
```

**Fig. 22** *Modelling of an add operation in LISA with carry flag generation as a side effect*

```
OPERATION ADD {
 DECLARE {
  GROUP src1, dst = { reg };
  GROUP src2 = { reg || imm };}
 SYNTAX { "add" dst "," src1 "," src2 }
 CODING { 0b0000 src1 src2 dst }
 SEMANTICS { _ADDI[_C] ( src1, src2 ) -> dst; }}

OPERATION reg {
 DECLARE {
  LABEL index; }
 SYNTAX { "R" index=#U4 }
 CODING { index=0bxxxx }
 SEMANTICS { _REGI(R[index])<0..31> }}
```

**Fig. 23** *Compiler semantics modelling of the add operation from Fig. 22 and a micro-operation for register file access (micro-operation '_REGI')*

a carry flag (and similar for other flags like zero or sign) in instructions like ADD does not vary between different target processors.

Frequently, there is no one-to-one correspondence between IR operations (compiler-dependent) and micro-operations (processor-dependent). Therefore, the code selector generator that works with the semantic sections must be capable of implementing complex IR patterns by sequences of micro-operations. For instance, it might be needed to implement a 32-bit ADD on a 16-bit processor by a sequence of an ADD followed by an ADD-with-carry. For this 'lowering', the code selector generator relies on an extensible default library of transformation rules. Contrarily, some LISA operations may have complex semantics (e.g. a DSP-like multiply-accumulate) that cover multiple IR operations at a time. These complex instructions are normally not needed for an operational compiler but should be utilised in order to optimise code quality. Therefore, the code selector generator analyses the LISA processor model for such instructions and automatically emits mapping rules for them.

The use of semantic sections in LISA enables a much higher degree of automation in code selector retargeting, since the user only has to provide the semantics per LISA operation, while mapping rule generation is completely automated (except for user interactions possibly required to extend the transformation rule library for a new target processor).

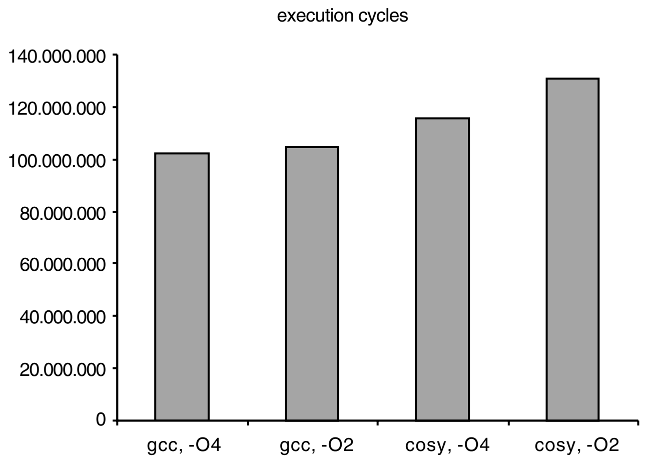
The semantics approach eliminates the above-mentioned model consistency problem at the expense of introducing a potential redundancy problem. This redundancy is due to the coexistence of separate behaviour (C/C++) and semantics (micro-operations) descriptions. The user has to ensure that behaviour and semantics do not contradict. However, this redundancy is easily to deal with in practice, since behaviour and semantics are local to each single LISA operation. Moreover, as outlined in [39], coexistence of both descriptions can even be avoided in some cases, since one can generate the behaviour from the semantics for certain applications.

## 5.4 Results

The retargetable LISATek C compiler has been applied to numerous different processor architectures, including RISC, VLIW and network processors. Most importantly, it has been possible to generate compilers for all architectures with limited effort, of the order of some man-weeks, dependent on the processor complexity. This indicates that the semi-automatic approach outlined in Section 5 works for a large variety of processor architectures commonly found in the domain of embedded systems.

While this flexibility is a must for retargetable compilers, code quality is an equally important goal. Experimental results confirm that the code quality is generally acceptable. Figure 24 shows a comparison between the gcc compiler (Section 3.2) and the CoSy based LISATek C compiler for a MIPS32 core and some benchmarks programs. The latter one is an 'out-of-the-box' compiler that was designed within two man-weeks, while the gcc compiler, due to its wide use, probably incorporates significantly more manpower. On average, the LISATek compiler shows an overhead of 10% in performance and 17% in code size. With specific compiler optimisations added to the generated backend, this gap could certainly be further narrowed.

Further results for a different target (Infineon PP32 network processor) show that the LISATek compiler generates better code (40% in performance, 10% in code



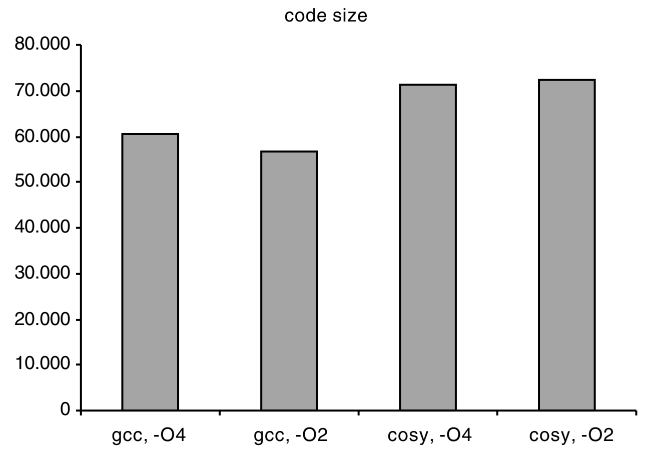


**Fig. 24** *Comparison between gcc compiler and CoSy based LISATek C compiler*

size) than a retargeted lcc compiler (Section 3.3), due to more built-in code optimisation techniques in the CoSy platform. Another data point is the ST200 VLIW processor, where the LISATek compiler has been compared to the ST Multiflow, a heavily optimising target-specific compiler. In this case, the measured overhead has been 73% in performance and 90% in code size, which is acceptable for an 'out-of-the-box' compiler that was designed with at least an order of magnitude less time than the Multiflow. Closing this code quality gap would require adding special optimisation techniques, e.g. in order to utilise predicated instructions, which are currently ignored during automatic compiler retargeting. Additional optimisation techniques are also expected to be required for highly irregular DSP architectures, where the classical backend techniques (Section 2.3) tend to produce unsatisfactory results. From our experience we conclude that such irregular architectures can hardly be handled in a completely retargetable fashion, but will mostly require custom optimisation engines for highest code quality. The LISATek/CoSy approach enables this by means of a modular, extensible compiler software architecture, naturally at the expense of an increased design effort.

## 6 Summary and outlook

Motivated by the growing use of ASIPs in embedded SoCs, retargetable compilers have made their way from academic research to EDA industry and application by system and semiconductor houses. While still being far from perfect, they increase design productivity and help to obtain better quality of results. The flexibility of today's retargetable compilers for embedded systems can be considered satisfactory, but more research is required on how to make code optimisation more retargetable.

We envision a pragmatic solution where optimisation techniques are coarsely classified with respect to different target processor families, e.g. RISCs, DSPs, NPUs and VLIWs, each of which show typical hardware characteristics and optimisation requirements. For instance, software pipelining and utilisation of SIMD (single instruction multiple data) instructions are mostly useful for VLIW architectures, while DSPs require address code optimisation and a closer coupling of different backend phases. Based on a target processor classification given by the user with respect to the above categories, an appropriate subset of optimisation techniques would be selected, each of which is retargetable only within its family of processors.

Apart from this, we expect growing research interest in the following areas of compiler-related EDA technology:

*Compilation for low power and energy:* Low power and/or low energy consumption have become primary design goals for embedded systems. As such systems are more and more dominated by software executed by programmable embedded processors, it is obvious that also compilers may play an important role, since they control the code efficiency. At first glance, it appears that program energy minimisation is identical to performance optimisation, assuming that power consumption is approximately constant over the execution time. However, this is only a rule-of-thumb, and the use of fine-grained instruction-level energy models [40, 41] shows that there can be a trade-off between the two optimisation goals, which can be explored with special code generation techniques. The effect is somewhat limited, though, when neglecting the memory subsystem, which is a major source of energy consumption in SoCs. More optimisation potential is offered by exploitation of small on-chip (scratchpad) memories, which can be treated as entirely compiler-controlled, energy efficient caches. Dedicated compiler techniques, such as [42, 43], are required to ensure an optimum use of scratchpads for program code and/or data segments.

*Source-level code optimisation:* In spite of powerful optimising code transformations at the IR or assembly level, the resulting code can be only as efficient as the source code passed to the compiler. For a given application algorithm, an infinite number of C code implementations exist, possibly each resulting in different code quality after compilation. For instance, downloadable reference C implementations of new algorithms are mostly optimised for readability rather than performance, and high-level design tools that generate C as an output format usually do not pay much attention to code quality. This motivates the need for code optimisations at the source level, e.g. C-to-C transformations, that complement the optimisations performed by the compiler, while retaining the program semantics. Moreover, such C-to-C transformations are inherently retargetable, since the entire compiler is used as a backend in this case. Techniques like [44–46] exploit the implementation space at the source level to significantly optimise code quality for certain applications, while tools like PowerEscape [47] focus on efficient exploitation of the memory hierarchy in order to minimise power consumption of C programs.

*Complex application-specific machine instructions:* Recent results in ASIP design automation indicate that a high performance gain is best achieved with complex application-specific instructions that go well beyond the classical custom instructions like multiply-accumulate for DSPs. While there are approaches to synthesising such custom instructions based on application code analysis [48–50], the interaction with compilers is not yet well understood. In particular, tedious manual rewriting of the source code is still required in many cases to make the compiler aware of new instructions. This slows down the ASIP design process considerably, and in an ideal environment the compiler would automatically exploit custom instruction set extensions. This will require generalisation of classical code selection techniques to cover more complex constructs like directed acyclic graphs or even entire program loops.

## 7 References

1 Aho, A.V., Sethi, R., and Ullman, J.D.: 'Compilers – principles, techniques, and tools' (Addison–Wesley, 1986)
2 Appel, A.W.: 'Modern compiler implementation in C' (Cambridge University Press, 1998)
3 Muchnik, S.S.: 'Advanced compiler design & implementation' (Morgan Kaufmann Publishers, 1997)
4 Wilhelm, R., and Maurer, D.: 'Compiler design' (Addison–Wesley, 1995)
5 Mason, T., and Brown, D.: 'lex & yacc' (O'Reilly & Associates, 1991)
6 Bischoff, K. M.: 'Design, implementation, use, and evaluation of Ox: an attribute-grammar compiling system based on Yacc, Lex, and C', Technical Report 92–31, Dept. of Computer Science, Iowa State University, USA, 1992
7 Leupers, R., Wahlen, O., Hohenauer, M., Kogel, T., and Marwedel, P.: 'An executable intermediate representation for retargetable compilation and high-level code optimization'. Int. Workshop on Systems, Architectures, Modeling, and Simulation (SAMOS), Samos, Greece, July 2003
8 Aho, A., Johnson, S., and Ullman, J.: 'Code generation for expressions with common subexpressions', *J. ACM*, 1977, **24**, (1)
9 Fraser, C.W., Hanson, D.R., and Proebsting, T.A.: 'Engineering a simple, efficient code-generator generator', *ACM Lett. Program. Lang. Syst.*, 1992, **1**, (3), pp. 213–226
10 Bashford, S., and Leupers, R.: 'Constraint driven code selection for fixed-point DSPs'. 36th Design Automation Conf. (DAC), 1999
11 Ertl, M. A.: 'Optimal code selection in DAGs'. ACM Symp. on Principles of Programming Languages (POPL), 1999
12 Chow, F., and Hennessy, J.: 'Register allocation by priority-based coloring', *SIGPLAN Not.*, 1984, **19**, (6)
13 Briggs, P.: 'Register allocation via graph coloring'. PhD thesis, Dept. of Computer Science, Rice University, Houston/TX, USA, 1992
14 Liao, S., Devadas, S., Keutzer, K., Tjiang, S., and Wang, A.: 'Storage assignment to decrease code size'. ACM SIGPLAN Conf. on Programming Language Design and Implementation (PLDI), 1995
15 Lam, M.: 'Software Pipelining: An Effective Scheduling Technique for VLIW machines'. ACM SIGPLAN Conf. on Programming Language Design and Implementation (PLDI), 1988
16 Wagner, J., and Leupers, R.: 'C compiler design for network processor', *IEEE Trans. Comput.-Aided Des. Integr. Circuits Syst.*, 2001, **20**, (11)
17 Wilson, T., Grewal, G., Halley, B., and Banerji, D.: 'An integrated approach to retargetable code generation'. 7th Int. Symp. on High-Level Synthesis (HLSS), 1994
18 Leupers, R., and Marwedel, P.: 'Retargetable compiler technology for embedded systems – tools and applications' (Kluwer Academic Publishers, 2001)
19 Marwedel, P., and Nowak, L: 'Verification of hardware descriptions by retargetable code generation'. 26th Design Automation Conf., 1989
20 Leupers, R., and Marwedel, P.: 'Retargetable code generation based on structural processor descriptions' (Kluwer Academic Publishers, 1998), Design Automation for Embedded Systems, vol. 3, no. 1
21 Leupers, R., and Marwedel, P.: 'A BDD-based frontend for retargetable compilers'. European Design & Test Conf. (ED & TC), 1995
22 Van Praet, J., Lanneer, D., Goossens, G., Geurts, W., and De Man, H.: 'A graph based processor model for retargetable code generation'. European Design and Test Conference (ED & TC), 1996
23 Target Compiler Technologies: http://www.retarget.com
24 Mishra, P., Dutt, N., and Nicolau, A.: 'Functional abstraction driven design space exploration of heterogenous programmable architectures'. Int. Symp. on System Synthesis (ISSS), 2001
25 Free Software Foundation/EGCS: http://gcc.gnu.org
26 Fraser, C., and Hanson, D.: 'A retargetable C compiler: design and implementation' (Benjamin/Cummings, 1995)
27 Fraser, C., and Hanson, D.: LCC home page, http://www.cs.princeton.edu/software/lcc
28 Associated Compiler Experts: http://www.ace.nl
29 Orailoglu, A., and Veidenbaum, A.: 'Application specific microprocessors (guest editors introduction)', *IEEE Des. Test Comput.*, 2003, **20**, pp. 6–7
30 Tensilica Inc.: http://www.tensilica.com
31 Stretch Inc.: http://www.stretchinc.com
32 Hoffman, A., Meyr, H., and Leupers, R.: 'Architecture exploration for embedded processors with LISA' (Kluwer Academic Publishers, 2002), ISBN 1-4020-7338-0
33 CoWare Inc.: http://www.coware.com
34 Scharwaechter, H., Kammler, D., Wieferink, A., Hohenauer, M., Karuri, K., Ceng, J., Leupers, R., Ascheid, G., and Meyr, H.: 'ASIP architecture exploration for efficient IPSec encryption: a case study'. Int. Workshop on Software and Compilers for Embedded Systems (SCOPES), 2004
35 Nohl, A., Braun, G., Schliebusch, O., Hoffman, A., Leupers, R., and Meyr, H.: 'A universal technique for fast and flexible instruction set simulation'. 39th Design Automation Conf. (DAC), New Orleans, LA, (USA), 2002
36 Wahlen, O., Hohenauer, M., Braun, G., Leupers, R., Ascheid, G., Meyr, H., and Nie, X.: 'Extraction of efficient instruction schedulers from cycle-true processor models'. 7th Int. Workshop on Software and Compilers for Embedded Systems (SCOPES), 2003
37 Hohenauer, M., Wahlen, O., Karuri, K., Scharwaechter, H., Kogel, T., Leupers, R., Ascheid, G., Meyr, H., Braun, G., and van Someren, H.: 'A methodology and tool suite for C compiler generation from ADL processor models'. Design Automation & Test in Europe (DATE), Paris, France, 2004
38 Ceng, J., Sheng, W., Hohenauer, M., Leupers, R., Ascheid, G., Meyr, H., Braun, G.: 'Modeling instruction semantics in ADL processor descriptions for C compiler retargeting'. Int. Workshop on Systems, Architectures, Modeling, and Simulation (SAMOS), 2004
39 Braun, G., Nohl, A., Sheng, W., Ceng, J., Hohenauer, M., Scharwaechter, H., Leupers, R., and Meyr, H.: 'A novel approach for flexible and consistent ADL-driven ASIP design'. 41st Design Automation Conf. (DAC), 2004
40 Lee, M., Tiwari, V., Malik, S., and Fujita, M.: 'Power analysis and minimization techniques for embedded DSP software', *IEEE Trans. Very Large Scale Integr. (VLSI) Syst.*, 1997, **5**, (2)
41 Steinke, S., Knauer, M., Wehmeyer, L., and Marwedel, P.: 'An accurate and fine grain instruction-level energy model supporting software optimizations'. Proc. PATMOS, 2001
42 Steinke, S., Grunwald, N., Wehmeyer, L., Banakar, R., Balakrishnan, M., and Marwedel, P.: 'Reducing energy consumption by dynamic copying of instructions onto onchip memory'. ISSS, 2002
43 Kandemir, M., Irwin, M.J., Chen, G., and Kolcu, I.: 'Banked scratch-pad memory management for reducing leakage energy consumption'. ICCAD, 2004
44 Falk, H., and Marwedel, P.: 'Control flow driven splitting of loop nests at the source code level'. Design Automation and Test in Europe (DATE), 2003
45 Liem, C., Paulin, P., and Jerraya, A.: 'Address calculation for retargetable compilation and exploration of instruction-set architectures'. 33rd Design Automation Conf. (DAC), 1996
46 Franke, B., and O'Boyle, M.: 'Compiler transformation of pointers to explicit array accesses in DSP applications'. Int. Conf. on Compiler Construction (CC), 2001
47 PowerEscape Inc.: http://www.powerescape.com
48 Sun, F., Ravi, S. *et al.*: 'Synthesis of custom processors based on extensible platforms', ICCAD 2002
49 Goodwin, D., and Petkov, D.: 'Automatic generation of application specific processors'. CASES 2003
50 Atasu, K., Pozzi, L., and Ienne, P.: 'Automatic application-specific instruction-set extensions under microarchitectural constraints'. DAC 2003