Microcode Generation for Flexible Parallel Target Architectures

Rainer Leupers, Wolfgang Schenk, and Peter Marwedel

University of Dortmund, Department of Computer Science XII,
44221 Dortmund, Germany

Abstract: Advanced architectural features of microprocessors like instruction level parallelism and pipelined functional hardware units require code generation techniques beyond the scope of traditional compilers. Additionally, recent design styles in the area of digital signal processing pose a strong demand for retargetable compilation. This paper presents an approach to code generation based on netlist descriptions of the target processor. The basic features of the MSSQ microcode compiler are outlined, and novel techniques for handling complex hardware modules and multi-cycle operations are presented.[1]

Keyword Codes: B.1.4
Keywords: Control Structures and Microprogramming, Microprogram Design Aids

1 Introduction

Besides instruction pipelining, two important means for increasing the throughput of microprocessors have been identified by hardware designers: instruction level parallelism and data pipelining. Instruction level parallelism comprises several functional units working independently from each other, typically in combination with a VLIW type controller, whereas the latter is often used in digital signal processors (DSPs) for accelerating multiply-accumulate sequences. Exploiting these advanced architectural features poses new challenges for compiler technology, since there is no longer a clear compiler/architecture interface via an instruction set. Furthermore, recent processor design styles in the DSP area established a new view of the role of compilers in the design process. The use of *application-specific instruction set processors* (ASIPs) provides a convenient compromise between pure hardware implementations (ASICs) and pure software solutions via programmable off-the-shelf processors [1]. Usually, ASIP architectures are not fixed, but are subject to change during the design process. This implies "moving" pieces of functionality between hardware and software, which in turn requires frequent re-mapping of the system behavioral description onto the target architecture for performance evaluation. In order facilitate compilation onto different targets, the code generation process should be *retargetable*, i.e. no manual compiler adaption should be necessary. We propose retargetability based on *pure structural target descriptions* at the register transfer level (fig. 1). The advantages of this approach are manifold:

1) A RT level netlist of the target structure is usually available during the system design process.

2) Code generation is based on a model given in an easy-to-learn hardware description language. Thus, the concept "naturally" fits into a CAD system for design automation.

3) The controller structure is part of the architectural model, therefore restrictions due to encoding or sharing are detected by the compiler, and code generation is not restricted to VLIW type controllers.

4) With the controller structure being part of the model, it is possible to map resource conflicts to instruction conflicts, facilitating the scheduling phase.

5) Changes within the target architecture are easily reflected by adapting the architectural model.

6) No re-compilation of the compiler itself is required when moving to a new target architecture.

In this paper we present retargetable compilation techniques based on RT-level netlist models, which are capable of exploiting instruction level parallelism as well as data pipelining. Binary machine code is generated for *predefined* structures, in contrast to synthesis systems like CATHEDRAL [2] and CAPSYS [3], that perform binary code generation for *automatically synthesized* structures, which is a less expendable task. The CodeSyn compiler by Paulin [1] presents another approach to retargetable code generation for predefined structures which is based on

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

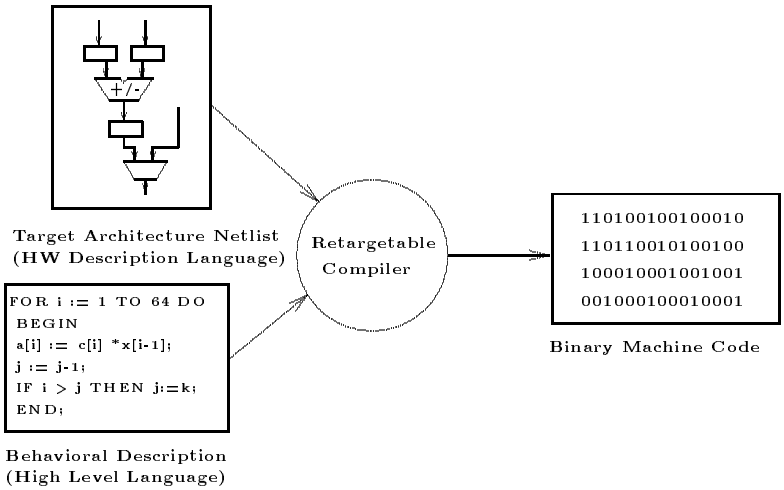

Figure 1: Retargetable compilation based on structural descriptions

specification of data flow patterns within the target hardware. However, these data flow patterns have still to be entered manually.

The paper is organized as follows. Sections 2 and 3 describe the modelling of architecture and behavior using the MIMOLA language. The basic steps for retargetable code generation (preallocation, pattern matching and scheduling/compaction) are presented in sections 4 to 6. These techniques have been implemented within the MSSQ compiler, which is part of the MIMOLA Design System [4]. Several restrictions of MSSQ have now been eliminated, e.g. pipelined modules and residual control are supported in the current version. These novel features are described in section 7. The paper ends with practical results and a conclusion.

2 Architectural models

The target architecture is modelled as a netlist on the register transfer level based on the MIMOLA language[2] with a PASCAL-like syntax. RTL modules are defined by their behavior based on a large set of primitive operations (arithmetic, logic, comparison, bit manipulation).

2.1 Combinational and sequential modules

Modules performing multiple operations are assumed to have a distinguished control input, e.g. a 16 bit ALU could be specified as follows, similar to a PASCAL procedure:

```
MODULE ALU (IN in1,in2:(15:0); OUT res:(15:0); FCT ctr:(1:0));
BEGIN
 res <- CASE ctr OF
         %00: in1 + in2;
         %01: in1 - in2;
         %10: in1 "AND" in2;
         %11: in1;
        END_CASE;
END;
```

Depending on the value of the control input `ctr`, the ALU either computes addition, subtraction or conjunction on the two data inputs `in1` and `in2` or passes `in1` to the output `res`. The data types are given as bitstrings in the

[2]See [5] for the complete syntax. Convertors from VHDL to MIMOLA are available, but we prefer the latter throughout this paper for sake of better readability.

format (<highbit>:<lowbit>). A 32 bit register with enabling signal and storing data at the rising clock edge is modelled by

```
MODULE Reg32bit(IN data:(31:0); OUT output:(31:0); FCT enable:(0); CLK clock:(0));
CONBEGIN
 CASE enable OF
  %0: "NOLOAD";                          (* do not load         *)
  %1: AT clock UP DO Reg32bit := data;   (* load at rising edge *)
 END_CASE;
output <- Reg32bit;                      (* read always         *)
CONEND;
```

The CONBEGIN ... CONEND construct denotes concurrent execution, in this case the register is always readable and concurrently stores input data at the rising edge when enable = 1. Memory modules are modelled similarly, having an additional address input. Modelling and code generation for multiport memories is provided.

2.2 Connections

Module interconnections are specified as a list of source and sink ports:

```
CONNECTIONS
  ALU.res          -> accumulator.input;
  accumulator.out  -> ALU.in1;
```

Bit subranges of modules ports may be referenced explicitly. Busses require a separate declaration due to their impact on the code generation process, i.e. the need for tristate operations of bus drivers:

```
BUS databus: (15:0);
```

2.3 Controller model

MSSQ was designed for code generation for microprogrammed structures. The underlying generic controller structure is depicted in fig. 2. One distinguished memory module has to be marked as the instruction memory and one

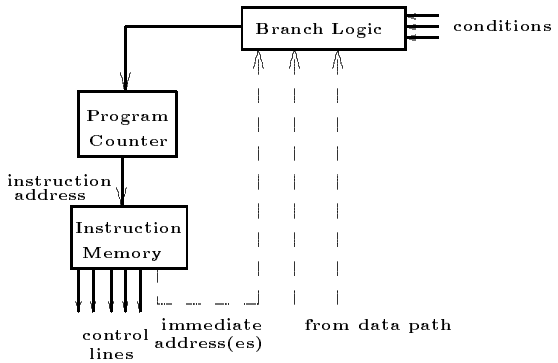

Figure 2: Generic controller structure

register as the program counter. The next program address is determined by an arbitrary branch logic possibly dependent on several condition codes. Five versions of control flow are considered during code generation:

1) **increment program counter:** The program counter is set to the following program address.

2) **unconditional jump:** Continue at a certain program address.

3) **then-branch:** Branch if a condition is true, otherwise increment program counter.

4) **else-branch:** Branch if a condition is false, otherwise increment program counter.

5) **two-way branch:** Branch to address a_1 if a condition is true, otherwise branch to address a_2.

2.4 High-level transformations

Besides the netlist model comprising modules and interconnections, MIMOLA permits description of *replacement rules*, i.e. correctness-preserving transformations of operations. Such replacement rules allow compilation of operations that are not directly supported by the hardware. Possible replacements include

```
REPLACE &a * 2 WITH &a + &a END;
REPLACE &a * 4 WITH "SHIFTL"(&a,2) END;
```

where `&a` denotes a formal parameter of the replacement rule. Replacement rules may be unconditional (i.e. are always applied) or conditional (i.e. the compiler decides on demand whether or not to apply the rule). A set of standard rules, e.g. for replacing high-level language constructs like FOR or WHILE loops by conditional jumps are provided in a library. Other replacements may be specified by the user.

3 Behavioral models

MIMOLA is a unified language for describing both structure and behavior. Behavioral descriptions in MIMOLA are essentially PASCAL programs. Several deviations exist regarding the allowed data types. MIMOLA programs permit bit-level addressing, direct access to hardware storages and calling hardware modules like procedures. Therefore, behavioral descriptions can be specified at different levels of abstraction, for instance the following two programs are valid and equivalent:

```
PROGRAM AtHighLevel IS
TYPE Integer = (15:0);
VAR a,b,c: Integer;
BEGIN
 a := 3 * b;
 c := a;
END;
```

```
PROGRAM AtRTLevel IS
BEGIN
 DataRAM[0] := 3 * DataRAM[1];
 accu := DataRAM[0];
END;
```

In the latter, the variables `a` and `b` are assumed to be located at cells 0 and 1 of memory `DataRAM`, and variable `c` has been physically mapped to register `accu`.

4 Preprocessing and preallocation

Several preprocessing steps are applied to the behavioral description.

1) Abstract user variables are mapped to physical memory locations.
2) High-level control structures (WHILE, FOR, REPEAT, ...) are replaced by equivalent conditional jump constructs. Only IF-statements may remain as control structures.
3) Unconditional replacement rules are applied.
4) Different implementations of remaining IF-statements are considered. This feature permits **extension of basic blocks** and thereby higher degrees of freedom for the microcode compaction phase. See [6] for an exhaustive discussion of IF-statement implementation.

During the *preallocation phase*, the *Connection Operation Graph* (COG) is constructed that represents the hardware structure. Vertices correspond to modules, and edges represent interconnections. Semantical knowledge about module operators is exploited by performing several local transformations within the COG. This includes entering additional paths for commutative operations and *via* operations. Via operations can be used for propagating values from a module input to the output using neutrals. When an ALU, for instance, can perform addition on the inputs i_1 and i_2, it implicitly has a via operation on each of the two inputs by setting the other one to zero. Allocation of via operations provides higher flexibility for data routing during code generation.

Analyzing the COG yields a set of *assertions*, i.e. necessary control codes that force modules to perform certain operations. A partial COG for the example ALU of section 2.1 is shown in fig. 3, assertions are denoted by exclamation marks. Besides the COG, the result of the preallocation phase is a list of partial control word settings (*versions*), each able to force execution of a certain operation on a certain module. All different versions are kept in order to provide greater flexibility for the code generation phase.

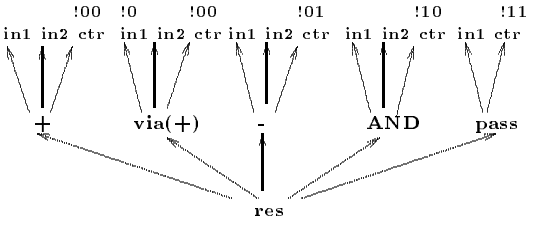

Figure 3: Partial Connection Operation Graph for an ALU

5 Pattern matching and allocation

After the preprocessing phase, the behavioral description to be mapped onto the target hardware consists of RT-level assignments. Assignment allocation in MSSQ is based on **matching dataflow patterns with subgraphs of the COG**.

5.1 Allocation of simple assignments

Considering the assignment `accu := DataRAM[0] + 17` the following subtasks have to be performed:

1) Enable `accu` for loading data
2) Provide address 0 at `DataRAM`
3) If necessary, set `DataRAM` to a readable mode
4) Allocate the constant 17
5) Allocate the addition operation on an ALU
6) Route the operands `DataRAM[0]` and 17 through the circuit to the ALU inputs
7) Route the result to the target `accu`

The COG is traversed in order to find the required operators, in this case addition. Providing the necessary control codes and constants relies on the results of the preallocation phase. Data routing is based on the COG interconnect structure and may require additional control codes, e.g. when exploiting via operations. If all subtasks can be solved, the assignment is finally tranformed into a set of partial control word settings, concurrently necessary to execute the assignment. If allocation fails, MSSQ generates an error message indicating the failure reasons, e.g. missing operators or data routes.

5.2 Sequentialisation

Assignments containing complex expressions like

```
regfile[2] := (DataRAM[1] * accu) + (regfile[1] SHIFTL 2);
```

in general require to be sequentialised. In this case, MSSQ relies on a list of distinguished possible temporary locations specified in the MIMOLA input and computes possible sequential versions. The resulting sequential assignments are treated as simple assignments.

5.3 Conditional assignments

After replacing high-level control structures in the preprocessing phase, assignments still may contain IF-statements, e.g.

```
IF cond THEN accu := DataRAM[0];
```

for which various implementations exist. Currently, two versions are implemented in MSSQ:

Conditional jump versions The above example can be transformed into the sequence

```
        IF cond THEN PC := PC + 1 ELSE PC := <label>
        accu := DataRAM[0];
<label>: <next instruction>
```

only requiring a multiplexer at the PC input. The assignment can be treated as a simple assignment.

Conditional load versions require a hardware structure as depicted in fig. 4, which often occurs in real microprocessors. Depending on a condition bit, the target storage module is either enabled or disabled. The data

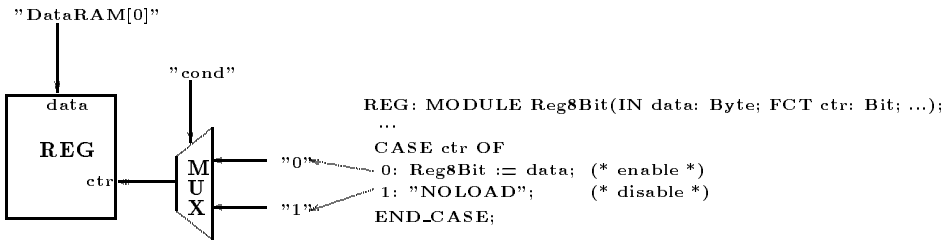

Figure 4: Hardware for conditional load operations

to be conditionally loaded is unconditionally routed to the storage data input, and the statement can remain unchanged. MSSQ tries to allocate both implementation versions. The better alternative is selected during microcode compaction.

6 Scheduling and compaction

The MSSQ scheduler heuristically tries to pack as many microoperations as possible into one machine instruction within each basic block. Since **resource conflicts are mapped to instruction conflicts**, it is sufficient to check whether the corresponding partial instructions are bit-compatible. Since all versions for execution of each statement are kept during the allocation phase, in case of incompatible operations the scheduler may select from several alternatives, and the shortest possible instruction sequence for each basic block can be selected. In addition to scheduling allocated assignments, any compiler based on structural hardware descriptions rather than on instruction sets has to **prevent undesired side effects** for each machine cycle. Two sources of side-effects must be taken into account:

Unused storage modules have to be disabled within each microinstruction in order to preserve their current state.

Unused bus drivers have to be disconnected by allocating tristate modes in order to avoid bus conflicts.

In general, additional control codes have to be supplied to prevent these side-effects. If a side-effect cannot be prevented, e.g. due to a missing register enable line, compaction fails and an error message is generated. The final result of the scheduling/compaction phase is a binary microprogram which realizes the specified behavior on the given target architecture.

7 Extensions for complex modules

MSSQ lacks from generating code for hardware structures with complex modules, such as pipelined modules, multiple cycle operations or multiple output operators. There are also restrictions imposed by the book-keeping of temporaries, which prevent the support of residually controlled modules.

We have overcome the deficiencies by developing a new approach, that accounts for complex modules, and has an improved data routing and book-keeping mechanism.

7.1 Complex module classification

There is a number of complex modules present in contemporary processors. They can be classified for code generation purposes – depending on the required control scheme – as described in the following.

Multiple cycle operations with fixed control are usually present at slow operators whose delay exceeds the cycle time. The code generator has to provide the control code stable during all cycles. The assumption, that each operation yields the result after an a priori known fixed number of cycles (the delay) is made.

Multiple cycle operations with initial control require the control code in the first cycle of an operation only. The operation takes multiple cycles to complete but needs no further control to do so.

Multiple cycle operations with variant control occur at programmable modules. The desired operation is decomposed into a sequence of more basic operations, each of which with a specific control code. The code generator emits code for each basic operation.

7.2 Residual control

A register is connected to a control input of the module. The code generator is therefore forced to load the desired control code into the register before the operation can be carried out. Loading the control code may be done one or more cycles in advance, but it must not be destroyed by an intermediate instruction.

7.3 Modeling complex modules

Whereas a key feature in the approach developed with MSSQ is the modeling of resource conflicts as instruction conflicts, the new approach deals with a redefined notion of resources, that suits the need for tracking the hardware resource usages over an interval of cycles. A *resource* is a register, a memory cell, a signal or an instruction field. A resource may be occupied by a value in a specific number of cycles. A *resource usage* is represented as a triple (r, v, i), where r is a resource, v is a value, and i denotes an interval of cycles. The following pipelined ALU latches all inputs in each cycle. It is of the initial control type.

```
MODULE MulDiv(IN a,b,c: int; OUT o: long);
VAR la,lb,lc: int;
CONBEGIN
 la := a; lb := b; lc := c;
 CASE lc OF
  0: o <- la * lb;
  1: o <- la / lb;
 END
CONEND;
```

The resources of the module are the signals a,b,c,o and the latches la,lb,lc. The latches are unconditionally loaded within each cycle. Such carriers are called *pipeline registers*. The behavior analysis extracts sets of resource usages for each path from a data source to a data sink. A path may include several pipeline registers, since pipeline registers are not regarded as data sink. The book-keeping mechanism keeps track of the variable *bindings* as well as the *machine state*. The machine state is a mapping of the resources to the values. Variables carried at a resource are said to be bound to it.

7.4 Code generation

A set of resource usages may contain one or more outputs of an operation, a set of side effects of the operation and the set of prerequisites (or assertions). A *template* is the partition of a set of resource usages into A, S and R. The elements of A, S, R are called *assertions*, *side effects* and *results* respectively. An operation is allocated when all prerequisites are allocated. If the resource denotes an instruction field, the value is a partial code version for the operation. If the resource refers to a register or storage cell, the book-keeping mechanism is considered whether or not the required value is already present. The set of side effects is used to update the book-keeping of the current machine state. Allocation, data routing and compaction can influence each other. During allocation

of a statement, the allocator collects partial code versions. The code generator checks for resource conflicts. Two resource usages $u_1 = (r_1, v_1, i_1), u_2 = (r_2, v_2, i_2)$ are *conflicting*, if $r_1 = r_2$ and $v_1 \neq v_2$ and $i_1 \cap i_2 \neq \emptyset$.

Whenever a (non pipeline) register has to be used as a temporary, the resulting partial code is tentatively compacted. If there is no valid schedule, because resource contention is exposed, a backtracking step is initiated. This scheme results in an exhaustive search for data routes.

With these extensions we expect a more versatile tool for a broad range of target architectures. The backtracking approach in allocation, data routing and compaction explores all versions for implementing a given basic block. Thus we can trade off the quality of the generated code against the time spent in searching for alternative versions.

8 Results

The MSSQ microcode compiler has been implemented by a total of about 34,000 PASCAL code lines and has been applied to a variety of real-life designs. These include:

- Verification of the SAMP processor [7].
- Code generation for the PROLOG processor PRIPS, which was recently fabricated through EUROCHIP.
- Assembly code generation for TI's TMS320C25 DSP, based on a novel code generation methodology [8].

Typical compilation rates are between 10 and 100 instructions per second on a SUN SparcStation 10. This is acceptable for the intended application area, i.e. code generation for flexible target architectures instead of standard processors.

9 Conclusions

A feasible approach to code generation for flexible programmable target architectures was presented. Due to higher compilation times, retargetable compilers are not expected to replace target-specific compilers for standard processors. The main application area can be identified as code generation for ASIPs. According to Paulin [1], there is a clear trend towards ASIPs as a design style for DSP systems. Compiler retargetability facilitates the selection of an appropriate ASIP architecture that meets the given timing constraints. In addition, the trade-off between hardware and software implementations of particular functions is supported.

References

[1] C.Liem, T.C.May, P.G.Paulin: Instruction Set Matching and Selection for DSP and ASIP Code Generation, Proc. European Design and Test Conference (EDAC), 1994

[2] D. Lanneer, F. Catthoor, G. Goossens, et al.: Open-ended System for High-Level Synthesis of Flexible Signal Processors, Proc. European Conference on Design Automation (EDAC), 1990, pp. 272-276

[3] G. Menez, M. Auguin, F Boeri, C. Carriere: Contribution of Compilation Techniques to the Synthesis of Dedicated VLIW Architectures, IFIP Transactions on Architectures and Compilation Techniques for Fine and Medium Grain Parallelism (A-23), 1993, pp. 217-228

[4] P. Marwedel, W. Schenk: Cooperation of Synthesis, Retargetable Code Generation and Test Generation in the MIMOLA Software System, European Design and Test Conference (EDAC), 1993, pp. 63-69

[5] R. Jöhnk, P. Marwedel: MIMOLA Reference Manual V 3.45, Technical Report No. 470, available from: University of Dortmund, Dept. of Computer Science, 44221 Dortmund, Germany

[6] P. Marwedel: Implementation of IF-statements in the TODOS microarchitecture synthesis system, IFIP Trans. on Synthesis for Control Dominated Circuits (A-22), 1993, pp. 249-262

[7] L. Nowak: SAMP: A General Purpose Processor Based on a Self-Timed VLIW Structure, ACM Comp. Arch. News, Vol. 15, No. 4, 1987, pp. 32-39

[8] R. Leupers, W. Schenk, P.Marwedel: Retargetable Assembly Code Generation by Bootstrapping, Proc. 7th International Symposium on High Level Synthesis, 1994