Tree-Based Mapping of Algorithms to Predefined Structures *

Peter Marwedel
University of Dortmund, Informatik XII
D-44221 Dortmund, Germany
e-mail: marwedel@ls12.informatik.uni-dortmund.de

Abstract

Due to the need for fast design cycles and low production cost, programmable targets like DSP processors are becoming increasingly popular. Design planning, detailed design as well as updating such designs requires mapping existing algorithms onto these targets. Instead of writing target-specific mappers, we propose using retargetable mappers. The technique reported in this paper is based on pattern matching. Binary code is generated as a result of this matching process. This paper describes the techniques of our mapper MSSV and identifies areas for improvements. As a result, it shows that efficient handling of alternative mappings is crucial for an acceptable performance.

1 Introduction

For many years, research on high-level design tools was focused on high-level synthesis. High-level synthesis starts with a behavioral description and generates a structure with the same behavior. In most of the cases, the generated structure inplements just the given behavior. The limitation of that approach lies in its low flexibility. Even minor variations of the behavior require a complete redesign and remanufacturing. This limitation does not exist for programmable systems, containing processors (microprocessors, DSP processors, processor cores from a cell library). These processors can be extended by custom datapaths or memories in order to obtain the necessary throughput. In the following text, we will call these systems our *target structures* or just *targets*. With these targets, manufacturing cost is significantly reduced, possibly at the expense of lower speed. This reduction in speed can be eliminated by a proper selection of custom datapaths. This way, some parts of the behavior are implemented by custom datapaths, other parts by programmable processors. Obviously, this is an example of hardware/software codesign.

Programmable targets result in the need for tools which map existing algorithms onto these. Such tools are required to examine tradeoffs between different targets. Analysing these tradeoffs is an important design action during the early design phases. With these tools it is possible to "play" around with different architectural options. Standard high-level synthesis does not have this functionality, because it usually assumes some freedom for the implementation. In many cases however (including DSP), the user has a good knowledge about the range of reasonable implementations. Currently, he cannot try these out, because mapping his application onto possible structures just takes too long. Recently, Paulin (in a talk at the high-level synthesis workshop) has emphasized the effort spent on mapping applications onto programmable systems. Techniques for reducing this effort would have a major economical effect.

Simple assembler-like tools are hard to use for evaluating different targets, because the application must be rewritten for each target. Compilers are only available for frequently used targets. Writing compilers for every target is too costly, especially because several algorithmic languages are in use. The lack of such compilers is a severe bottleneck during design style selection and early cost/performance estimation.

The solution is a retargetable mapper. Such a tool can be used to compile algorithms into binary code for predefined structures by performing a pattern-matching of the two descriptions.

2 Related work

We are aware of two so-called *checkers of executability*, which do the same type of matching [1, 2]. However, they do not generate binary instructions and hence do not perform the function which we need.

Some high-level synthesis systems are are capable of accepting partially specified structures as additional input and some also generate binary instructions. Using algorithms which were designed to synthesize

*Reprint from latex-source. Copyright of original publication: IEEE Computer Society.

structures in order to generate code would at least be inefficient. What will be described in this paper requires methods different from high-level synthesis, even though some similarities exist!

There has been some research on retargetable compilers for standard machine languages containing a similar pattern matching [3, 4]. They require a rather complicated partially manual preprocessing and hence cannot be used if targets change frequently.

More adequate for the current problem is previous work on retargetable *microcode* compilers [5, 6, 7, 8, 9]. Of these, the compilers by Mueller and Vegdahl can only be used for infrequent remapping to new targets because they require labour-intensive complicated preprocessing. Baba's work is mainly oriented towards mainframe microprogramming and the associated complicated next-address logic. The grammar-based approaches by Evangelisti and Mavaddat have to cope with an inherent ambiguity and complexity and consequently no results for complex designs are available up to now .

The approach taken in the MSS, however, applies to the new situation immediately. It does not require complex manual preprocessing and is based on a *true structural hardware description*. Two compilers have been designed for the MSS:

- MSSV, [10] based on tree matching and
- MSSC, [11] using graph matching.

The purpose of this paper is to present the basic ideas of MSSV, because many of the design decisions for MSSC are based on the observations made for MSSV.

3 Representation of Structures

The target structure has to be represented in our intermediate language TREEMOLA. The structural subset of TREEMOLA can be generated from MIMOLA or VHDL. The structure basically is described as netlist. The behavioral description of the components has to identify performed operations and the control codes which are required to select an operation.

Example: Assume that we want to map to the structure shown in fig. 1. The structure contains memory `SH`, ALU `ALU`, accumulator register `AC`, multiplexer `BMUX`, decoder `DEC`, instruction counter `PC` and instruction memory `I`. Memory `SH` allows accessing two independent locations concurrently. Henceforth, we will call the set of ports (in the sense of VHDL) to access a location, a *port group*. The input to MSSV describes the nets of fig. 1 as well as the behavior of the components. Some part of this input is listed in fig. 2.

In MIMOLA, the default datatype is the bit vector.

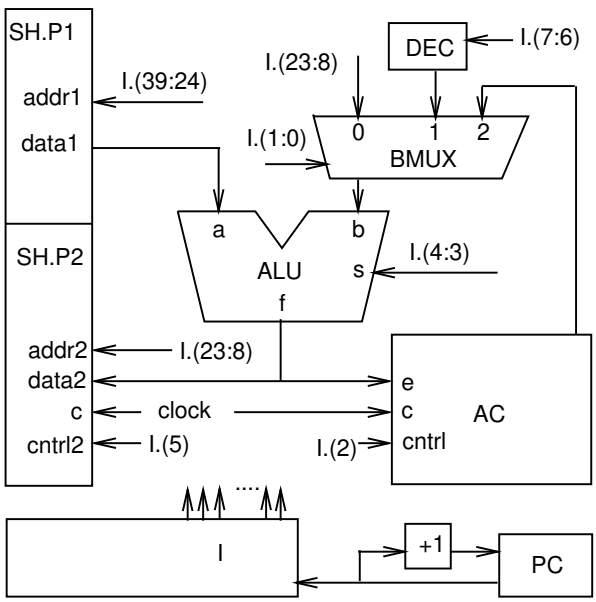

Figure 1: Target structure

```
TARGET simplecpu;
STRUCTURE IS PARTS              {components}
ALU: MODULE alu(a,b:(15:0);s:(1:0);
                OUT f:(15:0));
 BEHAVIOR b1 IS BEGIN
  f <-CASE s OF    {<- = signal assignment}
       0 : a * b;        1 : b ;
       2 : a  +  b;      3 : a  -  b;
 END; END;
DEC: MODULE dec(s:(1:0);OUT f:(15:0));
 BEHAVIOR b1 IS BEGIN
  f <- CASE s OF
       0: 0; 1:3; 2: 5; 3: 8;
 END; END;
SH: MODULE sh (IN addr1,addr2,data2:(15:0);
 OUT data1:(15:0);IN cntrl2:(0);CLK c:(0));
 BEHAVIOR b1 IS
  CONBEGIN
   AT c DO CASE cntrl2 OF
     0: sh[addr2] := data2; (* LOAD   *)
     1: ;                   (* NOLOAD *)
   END;
   data1 <- sh[addr1]       (* READ   *)
  CONEND;      ...    (other components)
 CONNECTIONS
 SH.data1 -> ALU.a;       ... (all nets)
```

Figure 2: Description of structure (incomplete)

Its index range is denoted as (*high-bit*:*low-bit*). The body of component descriptions is restricted to the

forms shown in the example. Each case clause defines an available component *operation mode* and the required *control codes* (the case labels). With its control input `s` set to `1`, component `ALU` is in a *transparent mode*. For sequential components, the operation identifiers `LOAD, NOLOAD` and `READ` are generated automatically. In MSSV, operation modes are also represented as TREEMOLA trees, very much like the trees used to represent the algorithm (see below). An available report [10] describes the full input to MSSV exactly.

Usually, temporary locations are required in order to map an algorithm onto a certain hardware. A group of locations has to be correspondingly tagged. Tagging is also required for the locations which are available for the algorithm's variables, the binary instructions and the program counter (controller state register). The following is an example in MIMOLA V3.45:

```
LOCATIONS_For_Temporaries     AC;
LOCATIONS_For_Variables       SH[0..9];
LOCATIONS_For_Instructions    I;{entire ROM}
LOCATIONS_For_ProgramCounter PC;
```

Subranges of bits of the output of the instruction memory are called *instruction fields*. In total, the structure is described by the netlist, the behavior of the components, the instruction fields, target-specific transformation rules (see below) and location lists.

4 Representation of the Algorithm

Our compilers also expect the algorithm to be described in TREEMOLA. Behavioral TREEMOLA can be generated from MIMOLA or other source languages. MIMOLA is essentially PASCAL, extended by bit-level addressing and language elements for describing hardware structures.

Example: As a (simple) running example, we will consider the assignment

```
a := b + 8
```

Source level text like this is first passed through standard tools MSSF, MSSR, and MSSI which perform among others the following functions:

- Translation from MIMOLA to TREEMOLA.
- Replacement of high-level language elements (e.g. loops and procedure calls) by simple assignments. Target-dependent, user-definable program transformation rules can be used to map procedure calls e.g. to pushes and pops of hardware stacks. Transformation rules are a mechanism for passing information about code generation for a particular target over to the compiler. Many changes of the target structure (like changes of the datapath) do not require changes to the rules.
- Replacement of variables by memory locations
- Replacement of unimplemented by implemented operators (e.g. replacement of `(a*2)` by `(a+a)`). This replacement is controlled by target-dependent, user-definable program rules.

After preprocessing, algorithms have been mapped to blocks of register transfers with the IF-statement being the only remaining high-level language element.

Example: The above assignment could be transformed into the TREEMOLA representation of the following statement:

```
SH[0] := SH[1] + 8
```

Preprocessing has replaced variables `a` and `b` by references to memory locations `SH[0]` and `SH[1]`.

MSSV uses the linked-node in-memory tree representation that is depicted in fig. 3. In that representation, `READ` and `LOAD` operations have been made explicit.

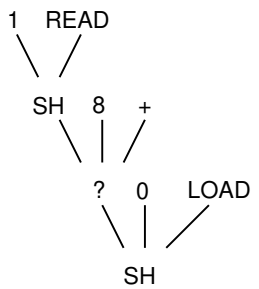

Figure 3: In-memory representation of an assignment

The meaning of a child depends upon its position:

- The rightmost child denotes an operation. The parent node can be interpreted as an `APPLY`-command for the operation and contains the name of the component, which should implement the operation (= '?' if no predefined operation/operator binding exists).
- The next child denotes an address, if the operation is either `READ` or `LOAD`.
- All other children denote data.

5 Matching Structure and Algorithm

5.1 Fundamental Concept

The main idea of the mapper is the following:

- For each constant of the algorithm, find decoders or instruction fields in order to generate it.
- For each operation of the algorithm, find ALUs with an appropriate operation mode.
- For each `READ` and `LOAD`-operation, find suitable memory ports.
- For each operation `e`, find paths from the modules computing the arguments to the input(s) of the modules implementing operation `e`.

During the matching process, MSSV attaches partial solutions to sections of the algorithm, for which matches have already been found. The matching process follows the 'recursive descend approach' which is used in traditional compilers. With this approach, the calling structure of code generation reflects the recursive nesting of syntactical elements. The outermost procedures handle blocks of assignments. For every assignment they call procedure `expression`.

5.2 Expressions

In the remaining procedures, variable `source` denotes a source (= an argument of an expression) and variable `sink` denotes the node that will be bound to the hardware performing the operation.
`expression` is the most important procedure:

```
PROCEDURE expression(e : tree);
 BEGIN
  FOR ALL source IN nodes(e)
  SEQUENCE right-to-left, depth-first DO
  CASE nodetype(source) OF
   IntegerTyp   : constant(source);
   OperationTyp : operation(e,source);
  END;
  IF NotLeaf(source) THEN bundling(source);
  IF source<>root(e) THEN path(source,sink);
 END;
```

This procedure traverses all nodes, starting with the leaves. On each level within the tree, it starts with the rightmost nodes, denoting operations to be performed. For these, `expression` calls procedure `operation`.

5.3 Matching of operations

In the example, the first node visited is the `LOAD`-node. `expression` calls `operation` to find components (or port-groups) with a matching operation mode. Matching includes the operation identifier, the number and bit-width of the arguments and the bit-width of the result. In order to select this operation mode, a control code must normally be applied to a control input. To generate this control code, `operation` calls `constant`. After calling `operation`, `expression` calls `path` to find a path from the component (or instruction field) generating the control code to the control input of the component corresponding to the sink node. The resulting information is called a *partial version*. Partial versions are attached to the sink node (see dashed line in fig. 4). `I(x).(y:z)` stands for a value `x` that should be supplied by instruction field `I.(y:z)`. Note that, in fig. 1, there is a direct path from the instruction field `I.(5)` to control input `SH.P2.cntrl2`.

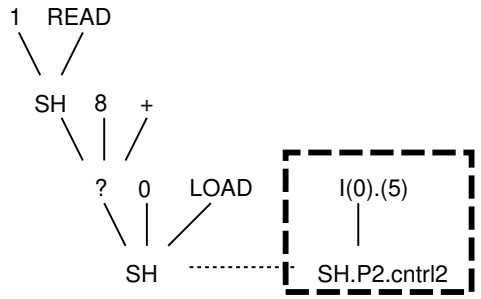

Figure 4: Partial version for `LOAD`-node

5.4 Matching of Constants

The node visited next by `expression` is the `0`-node. Another partial version is added to the tree (fig. 5).

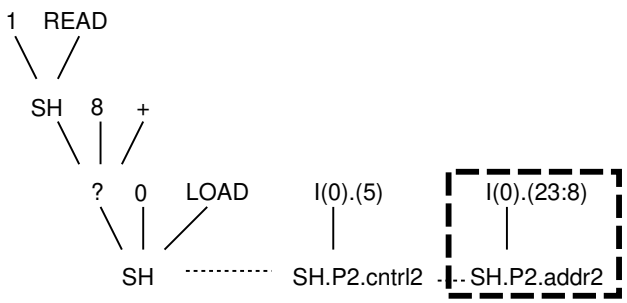

Figure 5: Partial version for `0`-node

This time the partial version describes how a `0` could be implemented at address inputs of components or port-groups for which partial versions for the control input exist. Constants like `0` can be generated by instruction fields, by hardwired constants and by decoders. The latter require control codes which in turn have to be generated by a call to `expression`.
The node visited next by `expression` is the `+`-node. Another partial version, describing the control code for `+` is linked to the TREEMOLA tree (see fig. 6).

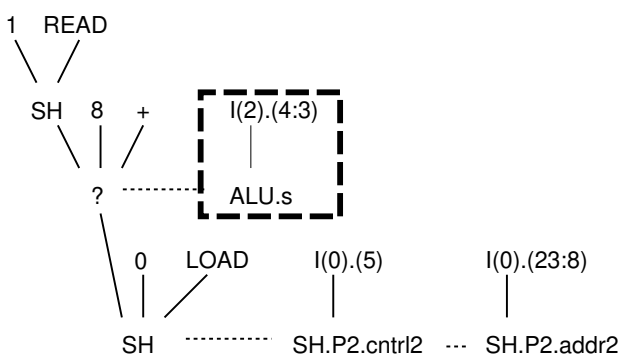

Figure 6: Partial version for `+`

Next, partial versions for the node containing the 8 will be generated. One version can be generated for the decoder and for the instruction field, respectively.

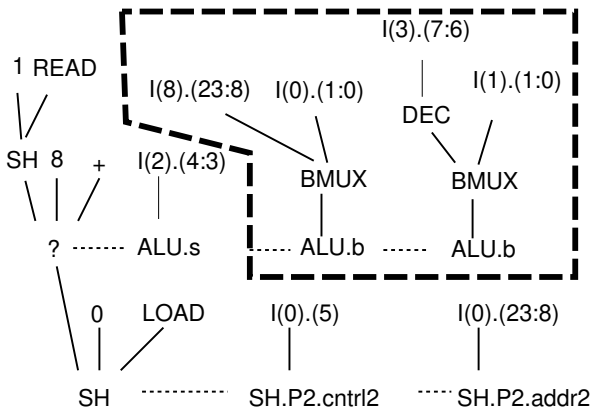

Figure 7: Partial versions for 8

5.5 Matching of Paths

In fig. 7, there is a reference to multiplexer `BMUX`. `BMUX` is required because there is no direct path from `I.(23:8)` and `DEC` to the sink's input and procedure `path` has to find a *via*. `path` calls procedures `dirpath`, `viapath` and `temppath`. `dirpath` scans the netlist to find a direct path from `source` to `sink`. `viapath` scans the netlist to find a path from `source` to vias. It adds a node describing the via to the tree and then calls `path` recursively to find a path from the via to candidates for the sink.

5.6 Insertion of Temporaries

`temppath` scans the netlist to find a path from `source` to a component containing temporary locations. If such a path is found, `temppath` creates an assignment of the tree with root `source` to that component. Furthermore, it adds a `READ`-operation to the remaining tree (with root `t`) and calls `expression` recursively for this tree. As a result of the insertion of temporary locations, *sequential versions* will be attached to root node `t`. Sequential versions consist of

1. an assignment to a temporary and the related non-sequential versions
2. the remaining tree with root `t`, for which this structure may be repeated.

Example: If `AC` is available as a temporary location, we could generate the sequence

```
AC := 8; (* via ALU.b *)
SH[0] := SH[1] + AC;
```

and the associated versions. This sequence would be required if the 8 were replaced by a constant which cannot be generated by `DEC`.

The nodes visited next therefore denote the `READ`-operation and the constant 1. The generated partial versions can be seen in fig. 8 and fig. 9, respectively.

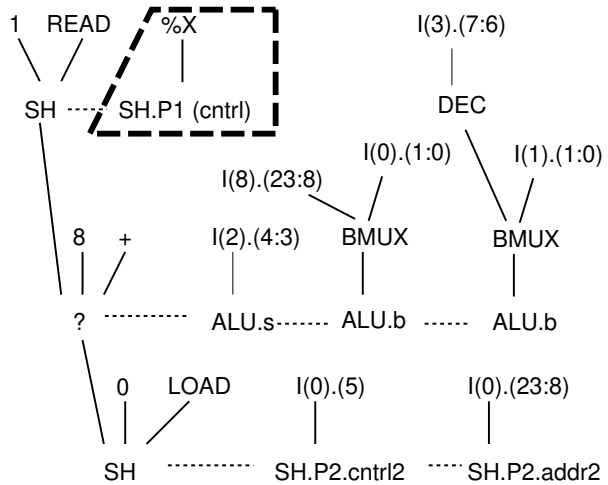

Figure 8: Partial version for `READ` (%X=don't care)

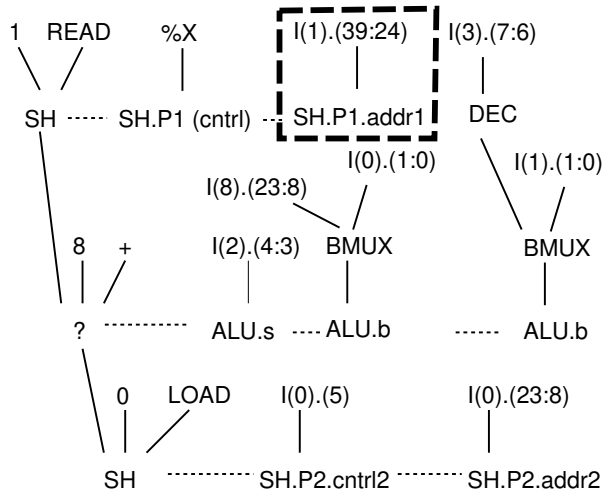

Figure 9: Partial version for 1

5.7 Bundling

Up till now, we have visited leaves. For these, `expression` does not call `bundling`. For other nodes, `bundling` tries to turn *partial versions* into *versions*. All possible combinations of partial versions are checked to find ways for implementing entire expressions in hardware. This task is called *bundling*. The result of the first call to `bundling` is shown in fig. 10. A version with root `SH.P1` is generated.

An important task of bundling is to detect resource conflicts: partial versions for the different input ports

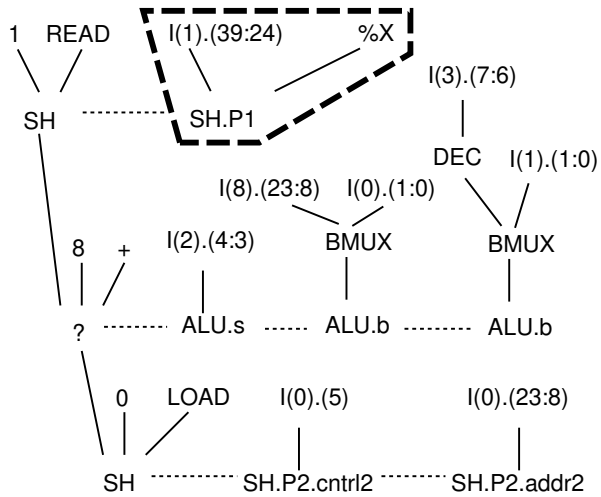

Figure 10: Result of bundling for node `SH`

could be in conflict to each other. Both MSSV and MSSC check for such conflicts by checking if there is a conflict at instructions. **Conflicts with respect to hardware components are mapped to conflicts at the control word (instruction conflicts). There is no blocking the hardware resources that are used in a certain control step.** Therefore, components which can perform several operations concurrently, can be modeled. This approach is feasible, because multiplexers are explicit in our hardware model. There is one exception, however: *programmable hardware conflicts*, i.e. *programmable bus conflicts*. This case is handled separately (see below). The call to `path` for `source = SH.P1` generates a partial version with root `ALU.a` (see fig. 11).

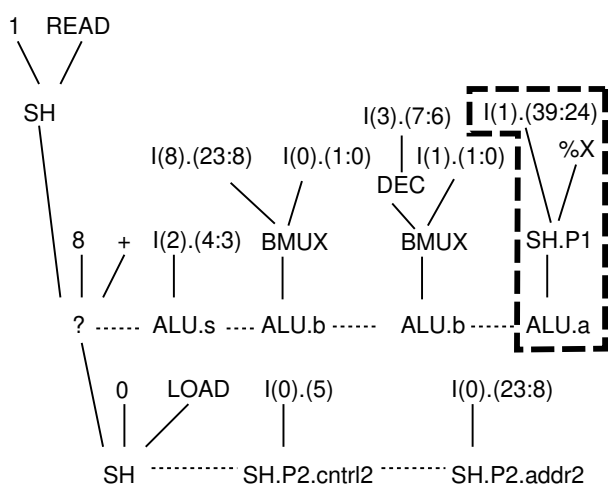

Figure 11: After calling `path` for `SH`

The outstanding completion of `expression` for the first two nodes results in two additional calls to `bundling` and `path`. This leads to figs. 12 and 13.

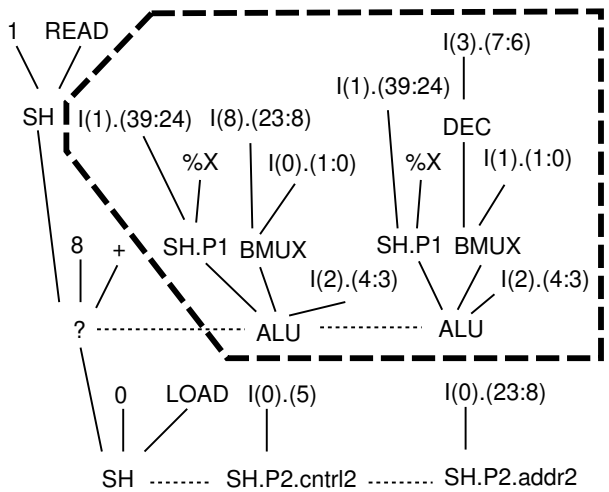

Figure 12: Result of bundling at `?`

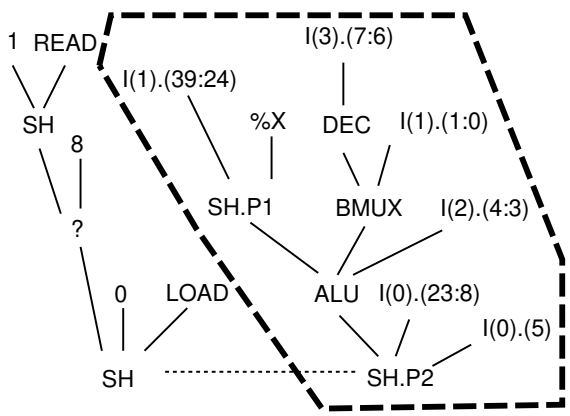

Figure 13: Result of bundling at 2nd `SH`-node

Note that the version containing `I(8).(23:8)` is not included in fig. 13 because it causes an instruction conflict with `I(0).(23:8)`.

6 Additional Tasks

6.1 Disabling buffers and storage devices

The code generated so far (as stored in the leaves of fig. 13) does not yet guarantee, that the states of sequential components, to which no assignment exists, remain unchanged. For example, the binary code which we just generated does not specify any value for the control input of `AC` (bit 2 of the instruction). We avoid such incorrect codes by generating versions for the `NOLOAD`-operation of sequential components.

Our scheduler augments the code generated so far by `NOLOAD`-versions for all components (or port groups) to which no assignment exists within a certain control step. In the case of our example, the scheduler adds a leaf to fig. 13, setting bit 2 to the code which disables `AC`. A similar method is used to set unused bus drivers into `TRISTATE`-mode.

6.2 Scheduling

Generated version lists are fed into the scheduler. The scheduler is an ASAP scheduler with provisions to handle alternate code sequences for each assignment. The scheduler tries to put as many statements into one control step as possible by checking available versions. The scheduler starts by trying to find a schedule with the smallest number of steps first.

6.3 Extraction of binary code

The binary code can be extracted from fig. 13 (augmented by `NOLOAD`-codes) quite easily (see table 1).

Bits	39:24	23:8	7:6	5	4:3	2	1:0
Code	1	0	3	0	2	1	1

Table 1: Binary code for the running example

7 Concluding remarks

7.1 Observations and limitations

The most important observation is that the number of versions for typical designs is significantly larger than expected. Hundreds of versions per statement are rather common for real designs. This slows down the code generation considerably. The reason for the large number of versions frequently originates from control code alternatives high up in the tree. If there are several such alternatives at various places in the tree, the cross product of these alternatives results in a huge amount of versions. Generating only one version at a time and then going to backtracking if it creates conflicts does not help, because we expect that the number of backtracks would be in the order of the number versions which MSSV creates. And backtracking would cause much more overhead.

Mapping as described above works for programmable targets with fixed length instructions. Multi-port memories and chaining are supported, but multi-cycle operations are not yet implemented. Due to its limited speed, the mapper cannot be used for large algorithms. Therefore, we are now using an extension.

7.2 Extension

Using the experience with MSSV, we have extended the mapping technique and implemented it in MSSC. MSSC uses the same formats for its input and output, but is faster. Selection of improvements is based upon observations made for MSSV.

- MSSC allows *version*-nodes (or-lists) anywhere in the tree, eliminating the cross-product problem.
- MSSC uses a more efficient method for handling some special transformation rules.

All full description of MSSC is beyond the scope of this paper. Although MSSC(uick) is faster than MSSV, MSSV is ideal for demonstrating the basic principles in an emerging discipline.

7.3 Results

First of all, the method that we have described above works for real architectures. We have tried it on various targets and it was able to generate code. A well-studied example is that of AMD-2900 based designs. They are challenging because of the strong encoding of instructions. Table 2 contains the size of the MIMOLA description of some standard components. It can be seen that the models are compact.

Circuit	Size of the HDL description	
	Lines	local components
AMD 2904	584	25
AMD 2910	414	14
AMD 29203	533	11
TMS 320C25	1995	95

Table 2: Size of target descriptions

Table 3 contains some code sequences, generated for our running example `simplecpu`. This code assumes that variables `a,b,c,d` are bound to locations `SH[0..3]`, respectively.

Source statements	Instructions	binary code (high-/low-bit)						
		39 24	23 8	7 6	5 5	4 3	2 2	1 0
a:=b+8	a :=b+8	1	0	3	0	2	1	1
a:=b+c	AC:=c-0	2	X	0	1	3	0	1
	a:=b+AC	1	0	X	0	2	1	2
a:=b+	AC:=d-0	3	X	0	1	3	0	1
(c+d)	AC:=c+AC	2	X	X	1	2	0	2
	a:=b+AC	1	0	X	0	2	1	2

Table 3: Generated code for some examples

The next example consists of the essential data computations of the differential equation solver benchmark, as specified by the following MIMOLA assignments:

```
x := dx + x;
u1:= u * dx;
y := y + u1;
u := u - ((u1 * (x * 5)) - (dx * (y * 3)));
```

Table 4 contains the corresponding code, using `simplecpu` as the target. It is assumed that variables `x, dx, u1, u, y` are bound to locations `SH[4..8]` and that locations `SH[10..12]` have been declared as additional temporary locations. Excluding setup times, MSSC needs 104 milliseconds to compile this example on an hp 9000/425 workstation. This corresponds to 134 instructions per second.

Source statements	Instructions	binary code (high-/low-bit)						
		39 24	23 8	7 6	5 5	4 3	2 2	1 0
diff_ eq_ lite	AC:=x-0	4	X	0	1	3	0	1
	x:=dx+AC	5	4	X	0	2	1	2
	AC:=dx-0	5	X	0	1	3	0	1
	u1:=u∗AC	7	6	X	0	0	1	2
	AC:=u1-0	6	X	0	1	3	0	1
	y:=y+AC	8	8	X	0	2	1	2
	h1:=x∗5	6	10	2	0	0	1	1
	AC:=u1-0	6	X	0	1	3	0	1
	h2:=h1∗AC	10	11	X	0	0	1	2
	h3:=y∗3	8	12	1	0	0	1	1
	AC:=dx-0	5	X	0	1	3	0	1
	AC:=h3∗AC	12	X	X	1	0	0	2
	AC:=h2-AC	11	X	X	1	3	0	2
	u:=u-AC	7	7	X	0	3	1	2

Table 4: Code for modified benchmark `diff_eq_lite`

7.4 Conclusion

We have described a method for generating binary machine code by matching true structural hardware description and the algorithm to be implemented. Such an approach, which does not need any instruction set description, is feasible for real architectures.

The importance of this approach reaches beyond the matching technique which we have described. The tool, which we have presented, has the potential of being the prototype of a whole new class of tools, which will link software and hardware. We predict that tools of this type will be essential for software/hardware codesign [12]. Furthermore, they will contribute to our understanding of the software/hardware interface. A better compiler technology for parallel architectures and superior CAD support for higher levels of abstraction could be results of this.

References

[1] F. Anceau. Force: A formal checker for executability. *in: D. Borrione (ed.): From HDL Descriptions to Guaranteed Correct Circuit Designs, Proc. of IFIP WG 10.2 Working Conf., North Holland*, 1986.

[2] S. Takagi. Rule based synthesis, verification and compensation of data paths. *Proc. IEEE Conf.Comp.Design (ICCD'84)*, pages 133–138, 1984.

[3] M. Ganapathi, C.N. Fisher, and J.L. Hennessy. Retargetable compiler code generation. *ACM Computing Surveys, Vol. 14*, pages 573–593, 1982.

[4] R. M. Stallman. Using and porting GNU CC. *Free Software Foundation*, 1993.

[5] C.J. Evangelisti, G. Goertzel, and H. Ofek. Using the dataflow analyzer on LCD descriptions of machines to generate control. *Proc. 4th Int. Workshop on Hardware Description Languages*, pages 109–115, 1979.

[6] T. Baba and H. Hagiwara. The MPG system: A machine-independent microprogram generator. *IEEE Trans. on Computers, Vol. 30*, pages 373–395, 1981.

[7] S.R. Vegdahl. Local code generation and compaction in optimizing microcode compilers. *PhD thesis and report CMUCS-82-153, Carnegie-Mellon University, Pittsburgh*, 1982.

[8] R.A. Mueller and J. Varghese. Flow graph machine models in microcode synthesis. *17th Ann. Workshop on Microprogramming (MICRO-17)*, pages 159–167, 1983.

[9] F. Mavaddat. Data-path synthesis as grammar inference. *IFIP-Workshop on Control Dominated Synthesis from A Register Transfer Description, Grenoble*, 1992.

[10] P. Marwedel. MSSV: Tree-based mapping of algorithms to predefined structures. Technical Report 431, Computer Science Dpt., University of Dortmund, 1993.

[11] P. Marwedel and L. Nowak. Verification of hardware descriptions by retargetable code generation. *26th Design Automation Conference*, pages 441–447, 1989.

[12] R. Leupers and W. Schenk. Retargetable assembly code generation by bootstrapping (extended version). Technical Report 488, Computer Science Dpt., University of Dortmund, 1993.