MSSV: Tree-Based Mapping of Algorithms to Predefined Structures (Extended Version)

Peter Marwedel
Lehrstuhl Informatik XII
University of Dortmund

Report No. 431

January 1993

Abstract

Due to the need for fast design cycles and low production cost, programmable circuits like FPGAs and DSP processors (henceforth called *target structures*) are becoming increasingly popular. Design planning, detailed design as well as updating such designs requires mapping existing algorithms onto these circuits. Instead of writing target-specific mappers, we propose using retargetable mappers. The technique reported is this paper is based on pattern matching. Binary code is generated as a result of this matching process. This paper describes the essential 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. This report is also intended as a reference for new developments.

Contents

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 implements just the given behavior. The disadvantage of that approach is its lack of flexibility. Even minor variations of the behavior require a complete redesign and even complete remanufacturing. This disadvantage can be eliminated by using programmable microprocessors, of-the-shelf DSP processors (e.g. [TI91]) or field programmable gate arrays (FPGAs) (see e.g. [XIL91]). With these targets, manufacturing cost is significantly reduced at the expense of lower speed. This reduction in speed can be reduced by mapping some parts of the behavior onto special, fixed structures and others onto programmable targets (hardware/software codesign) [IFI93]. This approach is used, for example, in CATHEDRAL-2nd [LCG$^+$90, GCLM92].

Programmable targets result in the need for tools which map existing applications (algorithms) onto these. 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 different languages for the description of algorithms are in use. The lack of such compilers is a severe bottleneck during design style selection and early cost/performance estimation[1].

The solution is a retargetable compiler. Such a tool can be used to map algorithms to structures, generating the required binary code by doing a pattern-matching between the two descriptions.

There has been some research on retargetable compilers for standard machine languages (see e.g. [GFH82]). They require a rather complicated partially manual preprocessing and hence cannot be used in a design automation environment.

More adequate for the current problem is previous work on retargetable microcode compilers [EGO79, BH81, Veg82, MV83, War89, Mav92]. 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 does not have this limitation, but it is mainly oriented towards mainframe microprogramming and the associated complicated next-address logik. The grammar-based approaches by Evangelisti and Mavaddat have to cope with an inherent ambiguity and complexity and consequently no results are available up to now for complex designs.

The approach taken in the MSS [Mar84, Now87], however, applies to the new situation immediately. It does not require complex manual preprocessing and is based on a *true structural hardware description* which is normally available in any design automation environment. If only functional models are available, structures with an equivalent behavior can be used instead. Such structures can be automatically generated from the functional model by using high-level synthesis. Using structural models solves many of the problems that compiler writers have with new pipelined superscalar machines. Explanation or description of the behavior of those machines is only possible, if structural components are identified. Problems, which we have been unable to solve previously (such as handling variable instruction wordlengths and visible multi-phase clocking elegantly), have disappeared with the new RISC machines.

Other attributes of targets within the microprogramming area became attributes of current

[1]The *Architect's Workbench* project [Fly90] is addressing this problem in a slightly different context.

processors (explicit parallelism, explicit modelling of the underlying structure, etc.) [Fis91].

Two compilers have been designed for the MSS:

- MSSV, based on tree matching and
- MSSQ [Now87, MN89], using graph matching. An extension of this is MAPS[2].

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

2 Representation of Structures

The target structure has to be represented in our intermediate language TREEMOLA (see [BPSJ91] for the latest version). TREEMOLA describes trees of nodes. The structural subset of TREEMOLA can be generated from MIMOLA or VHDL. The structure basically is described as a flat netlist. The behavioral description of the components (or cells) has to identify performed operations as well as the control codes which are required to select a specific operation.

Example:

Assume that the want to map to the structure shown in fig. 1.

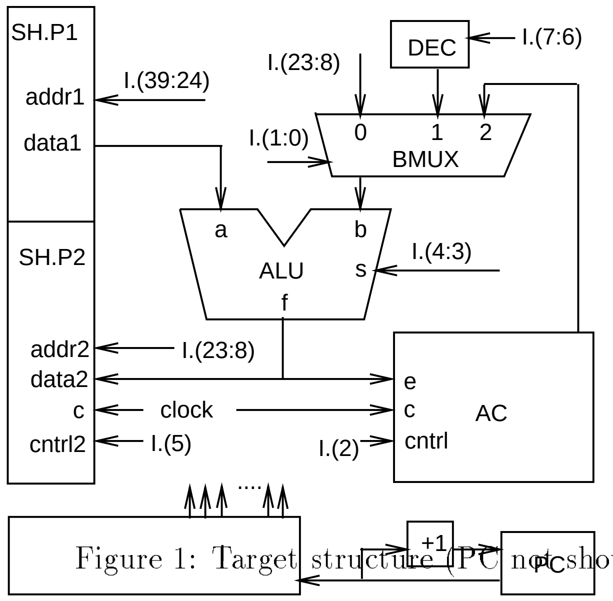

Figure 1: Target structure (PC not shown)

The structure contains two-port[3] memory `SH`, ALU `ALU`, accumulator register `ACCU`, multiplexer `BMUX`, decoder `DEC` and instruction memory `I`. The program counter is irrelevant for

[2]The (mapping algorithms to predefined structures)-compiler, which is currently being developed by W. Schenk.

[3]In order to solve the conflict between the use of the term 'port' in VHDL and its meaning in the context of multi-port memories, we will henceforth refer to the latter as *port-groups*. Thus, `P1` is a port-group, consisting of ports (in the sense of VHDL) `addr1` and `data1`.

the description in this paper.

The input to MSSV describes the nets of fig. 1 as well as the behavior of the components. Using MIMOLA V3.45 syntax, the full example will look like this:

```
  TARGET simplecpu;
  STRUCTURE IS
   PARTS
    SH: MODULE S64k
      PORT P1 (OUT data1:(15:0);ADR addr1:(15:0));
      PORT P2 (IN data2:(15:0);ADR addr2:(15:0);FCT cntrl2:(0);CLK c:(0));
      CONBEGIN
       WITH P1 DO
        data1 <- S64k[addr1];                (* <- = signal assignment *)
       WITH P2 DO
        CASE cntrl2 OF
         #1        : "NOLOAD" after 0;                      (* # = hex *)
         #0        : AT c UP DO S64k[addr2] := data2
        END;
      CONEND;
    I:MODULE SRAM
      (OUT data:(47:0);ADR addr:(15:0));
      BEGIN
       data <- SRAM[addr];
      END;
    ALU: MODULE Balu (OUT f:(15:0);IN a,b:(15:0);FCT s:(1:0));
      BEGIN
       f <- CASE s OF
         #3        : a "-" b;
         #2        : a "+" b;
         #1        : b;
         #0        : a
        END;
      END;
    PC: MODULE Rw(OUT a:(15:0);IN e:(15:0);CLK c:(0));
      BEGIN
       a <- Rw;
       AT c UP DO Rw := e;
      END;
    ACCU: MODULE REG(OUT a:(15:0);IN e:(15:0);FCT cntrl:(0);CLK c:(0));
      CONBEGIN
       a <- REG;
       CASE cntrl OF
        #1        : "NOLOAD" after 0;
        #0        : AT c UP DO REG := e
       END;
      CONEND;
    DEC: MODULE ADEC(OUT a:(15:0);FCT s:(1:0));
      BEGIN
       a <-CASE s OF
         #3: 8;  #2: 4; #1: 2;   #0: 0
        END;
      END;
    INC: MODULE Ai(OUT a:(15:0);IN e:(15:0));
      BEGIN
       a <- "INCR" e;
      END;
    Clock: MODULE Ck (OUT a:(0));
      BEGIN
       a <- "TOGGLE" up after 100 down after 100;
      END;
    BMUX: MODULE NMUX (OUT f:(15:0);IN a,b,c:(15:0);FCT s:(1:0));
      BEGIN
       f <- CASE s OF
         #3  : UNDEFINED; #2  : c; #1  : b; #0  : a
        END;
      END;

   CONNECTIONS
    Clock          -> SH.P2 .c;      ALU            -> ACCU  .e;
    Clock          -> ACCU  .c;      I.data .(2)    -> ACCU  .cntrl;
    Clock          -> PC    .c;      I.data .(31:16)-> BMUX  .a;
    I.data .(47:32)-> SH.P1 .addr1;  DEC            -> BMUX  .b;
    ALU            -> SH.P2 .data2;  ACCU           -> BMUX  .c;
    I.data .(31:16)-> SH.P2 .addr2;  I.data .(1:0)  -> BMUX  .s;
    I.data .(5)    -> SH.P2 .cntrl2; I.data .(7:6)  -> DEC   .s;
    SH.P1          -> ALU   .a;      INC            -> PC    .e;
    BMUX           -> ALU   .b;      PC             -> INC   .e;
    I.data .(4:3)  -> ALU   .s;      PC             -> I     .addr;
  END;
```

In MIMOLA, the default datatype is the bit vector. Its index range is denoted as (*high-bit*:*low-bit*). The body of component descriptions is restricted to the forms shown in the example. Note that for each case label we define 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).

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  ACCU;
LOCATIONS_For_Variables    SH[0..1000];
LOCATIONS_For_Instructions I;{entire ROM}
LOCATIONS_For_ProgramCounter PC;
```

MIMOLA allows the reservation of lists of components and locations for temporaries and for variables. Equivalent language elements are missing in VHDL'87. In VHDL'92, *groups*

[IEE92] can be used to represent lists of components. There is still no way to make reservations for locations within a component.

Def.: 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.

In order to speed up the mapping algorithm, some preprocessing of the hardware structure is performed. Computing the *reachability*-relation is the most important part of this.

Def.: Port i can be directly reached from port j, or $j \leadsto i$ iff

1. i is an input and j is an output and both belong to the same net.
2. j is a (data) input and i is an output of the same component and the component can be used as a temporary or as a via (mux, ALU with an transparent mode, ALU with an operation mode having a neutral element).

Def.: Relation $\leadsto^*$ is the transitive closure of relation $\leadsto$.

This relation is stored for every structure and reused if the structure is not changed.

3 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[4]. MIMOLA [BMSJ91] is essentially PASCAL, extended by bit-level addressing and language elements for describing hardware structures. The following language features are included:

- FOR, WHILE, REPEAT
- ARRAYS (any dimension)
- Procedures (any static level), formal parameters
- Inline functions (restricted form)
- Explicit bit-level addressing, FIELD-declaration
- Explicit parallelism (PARBEGIN)
- Operator set roughly equivalent to VHDL'92
- 2 logic values (4 for hardware description)
- References to hardware components allowed
  Examples:
  `+'ALU` or `+_ALU`
  `ALU(b,8,1)`
- Variables may be bound to components
  Example:
  `VAR sp AT SH[0];`

[4] A compiler for VHDL is currently being developed.

- Partial implementation of dynamic data structures

As a (very 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 for example 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 or other mechanisms for implementing procedures

  Example:

  ```
  MACRO ForProcedures IS
   REPLACE CALL &id WITH BEGIN save:=pc&; pc&:= &id END;
   REPLACE RETURN   WITH pc&:= save+1 END;
  END;
  ```

  These transformation rules can be used in case of parameter-less, unnested procedure calls. Calls to any procedure `&id` are replaced by two assignments. The first assignment saves the program counter in register `save`. The second stores the entry label `&id` in the program counter. All identifiers starting with an `&`-sign are formal parameters. All identifiers ending with an `&`-sign are built-in functions.
- Replacement of abstract variables by memory locations (variable-storage binding for the algorithm's variables).
- Computation of the index ranges for all bit vectors.
- Operator canonization: e.g. replacement of $<$ by $>$ and swapping of arguments for commutative operations such that constants will never be used as left arguments.
- Replacement of unimplemented operators by implemented operators (e.g. replacement of `(a * 2)` by `(a + a)`). This replacement is again controlled by target-dependent, user-definable program transformation rules.

This preprocessing basically has already been described elsewhere [Mar84]. 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 have been transformed into the following slightly simplified TREEMOLA syntax:

```
(LOAD'SH(+(READ'SH(1)|8)0)
```

Note that preprocessing has already replaced references to variables `a` and `b` by references to memory locations `SH[0]` and `SH[1]`. `READ` and `LOAD` operations for `SH` have been made explicit.

MSSV uses the linked-node in-memory TREEMOLA representation that is depicted in fig. 2.

In fig. 2, the meaning of a son depends upon its position:

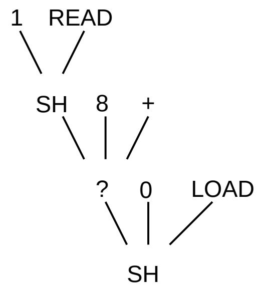

Figure 2: In-memory representation of an assignment

- The rightmost son denotes an operation. The father node (the *sink 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). This format for operators has been chosen because the operator nodes later will represent an additional input to the component, generating required control codes.
- The next son denotes an address, if the operation is either READ or LOAD.
- All other sons denote data.

4 Matching of Structures and Algorithms

4.1 Assignments and Expressions

The matching procedure follows the 'recursive descend approach' which is used in many traditional compilers. With this approach, the calling structure of code generation reflects the recursive nesting of expression trees. The outermost procedures handle blocks. They do not perform any matching and are therefore skipped in this report. The first procedure which we consider is procedure `assignment`. In our example, it will be called with (a pointer to) the root node as an argument. Procedure `assignment` basically just calls procedure `expression`:

```
PROCEDURE assignment(s : tree);
 BEGIN
  expression(s);
  (* other actions concern the generation*)
  (* of error messages, the output of    *)
  (* generated code and book-keeping for *)
  (* temporary locations                 *)
 END;
```

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

The most important procedure is procedure `expression`:

```
PROCEDURE expression(e : tree);
 BEGIN
  FOR ALL source IN nodes(e)
  SEQUENCE right-to-left, depth-first DO
  CASE nodetype(source) OF
   IntegerTyp, LabelTyp, StringTyp: constant(source);
   OperationTyp:                    operation(e,source);
   OTHERWISE   :;
  END;
  IF nodetype(source)=CatenationTyp
  THEN catbundling(source)
  ELSE
   BEGIN
    bundling(source);
    IF source<>root(e) THEN path(e,source,sink);
   END;
 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`. `operation` tries to find ways for implementing operations in hardware and links a description of the required control code to the expression tree. Furthermore, `expression` calls procedure `constant`, which tries to find ways for implementing constants in hardware. Possible implementations are linked to the constant itself. `catbundling`, `bundling` and `path` will be described below.

4.2 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. 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`.

The pseudo code for procedure `operation` is the following:

```
PROCEDURE operation(e, source : tree);
 BEGIN
  Let w be a component which is able to implement the operation denoted
  by source (matching includes the operation identifier and the number
  and bit-width of the arguments;
  commutativity and possible use of wider components for some operations
  is taken into account).
  IF implementation needs control code THEN
   call expression to generate this code;
  Link generated information to source;
 END;
```

Note that the search for matching operations is based on comparing operation identifiers. Overloading of operation identifiers, except for different bit vector lengths, is not possible.

The situation after the completion of the calls to `operation` and `constant` can be seen in fig. 3. The generated information is called a *partial version*. Partial versions are attached to the tree (see dashed line in fig. 3). `I(x).(y:z)` stands for a value `x` that should be supplied by instruction field `I.(y:z)`.

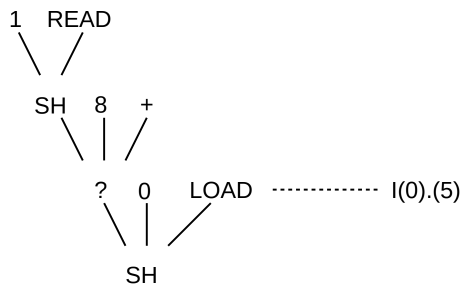

Figure 3: Situation after completion of `constant`

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. Note that, in fig. 1, there is a direct path from the instruction field to the control input.

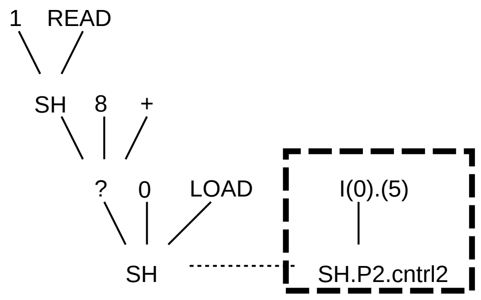

Figure 4: Situation after completion of `path`

4.3 Matching of Constants

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

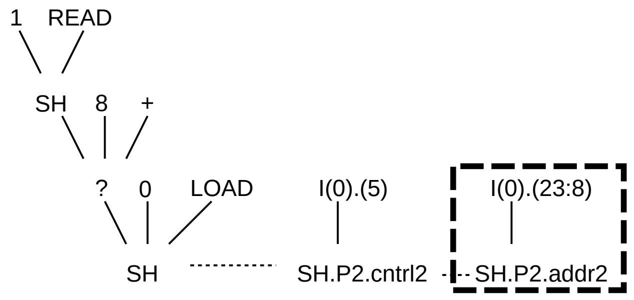

Figure 5: Partial version for `0`-node

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 matching between the algorithm and the structure considers the fact that constants can be generated by concatenations. Hence, the routine starts the matching at one end of the constant and accepts matches of subranges of the full bitwidth. In the case of such a partial match, it calls itself recursively for the remaining bits:

```
PROCEDURE constant(source : tree);
 BEGIN
  FOR ALL field IN instruction-fields DO
   IF length(field) <= length(source)
    THEN BEGIN
     Create partial version I(value) with length(field);
     Link version to source;
     IF length(field)<length(source) THEN
      call constant for remaining bits;
    END;
  FOR ALL components DO
   IF component is able to supply required constant value THEN
    BEGIN
     Create partial version, calling the component with the
     required control code.
     IF the control code is <> don't care
      THEN call expression to generate control code;
     IF component supplies only subrange of bits
      THEN call constant for remaining bits;
    END;
 END;
```

Procedure `constant` analyses reachability relation $\rightsquigarrow^*$ in order to ignore fields and components which cannot be connected to the sink.

The fact that constants can be generated by concatenations is responsible for a significant amount of execution time of MSSV. In order to reduce this time, MSSV supports a user-definable limit for the minimum number of bits per argument of the concatenation.

The node visited next by `expression` is the `+`-node. Another partial version, this time describing the control code for `+` is linked to the TREEMOLA tree (see fig. 6).

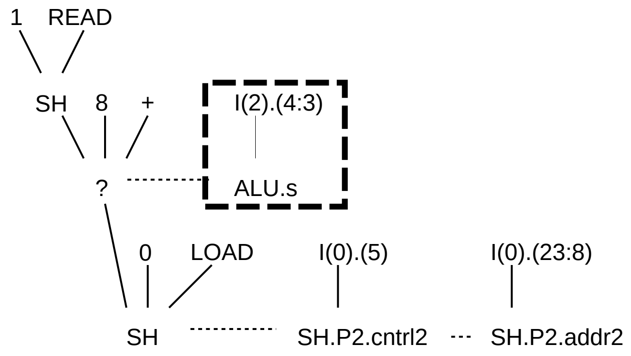

Figure 6: Partial version for `+`

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 (c. f. fig. 7).

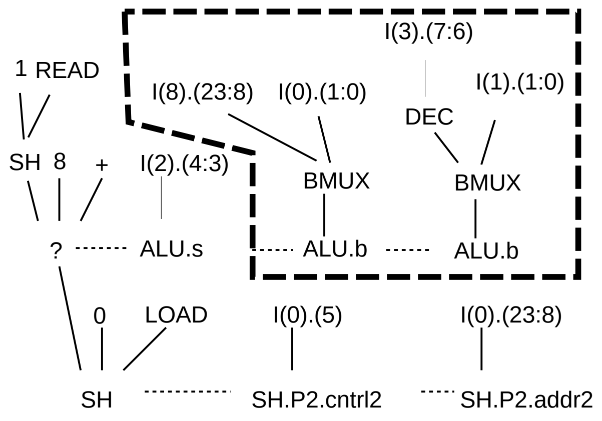

Figure 7: Partial versions for 8

4.4 Matching of Paths

In fig. 7, there is a reference to multiplexer BMUX. This multiplexer is required because there is no direct path from I.(31:16) and DEC to the sink's input and procedure path has to find a via through multiplexer BMUX. The pseudo code for path is the following:

```
PROCEDURE path(t, source, sink: tree);
 BEGIN
  dirpath(source,sink);
  IF additional paths are possible THEN
   BEGIN
    viapath(t, source, sink);
    temppath(t, source, sink);
   END;
 END;
```

Procedure path also analyses reachability relation $\rightsquigarrow^*$ in order to ignore fields and components which cannot be connected to the sink. 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.

4.5 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. This remaining tree is tagged such that the same component cannot be used again unless some useful operation has been performed on the data. 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 `ACCU` were available as a temporary location, we would generate

1. `ACCU := 8; (* via ALU.b *)` and the associated versions
2. `SH[0] := SH[1] + ACCU;` and the associated versions

This sequence would be required if the `8` would be replaced by a constant which cannot be generated by `DEC`. The nodes visited next by `expression` denote the `READ`-operation and the constant 1. The generated partial versions can be seen in fig. 8 and fig. 9, respectively.

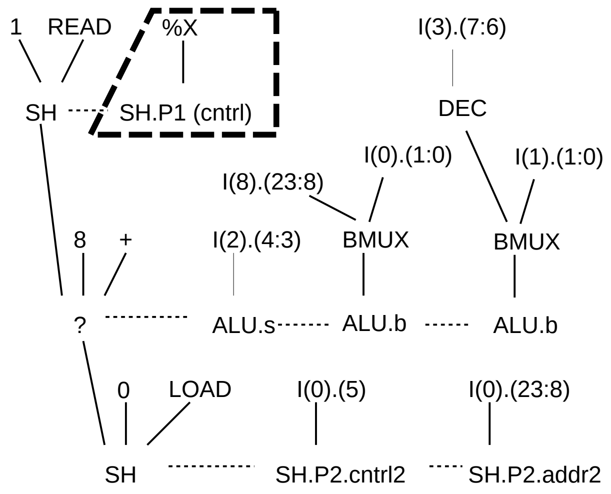

Figure 8: Partial version for `READ` (%X=don't care)

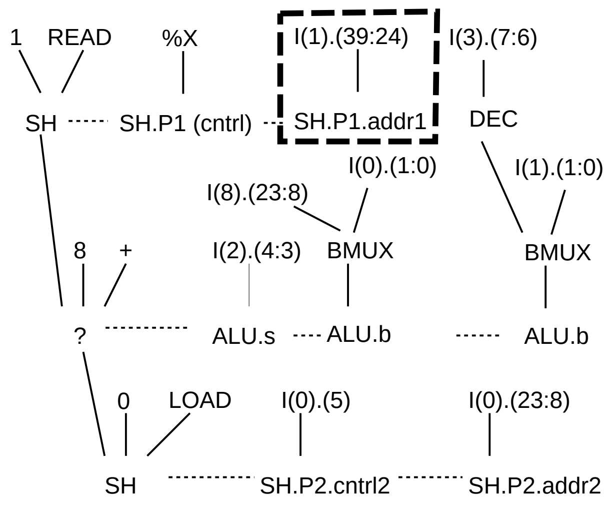

Figure 9: Partial version for `1`

4.6 Bundling

Up till now, we have visited only leaves. For these, the calls to `bundling` and to `catbundling` within `expression` have no effect. For the other nodes, these procedures will try to turn *partial versions* into *versions*. All possible combinations of partial versions are checked in order to find ways for implementing an entire expression in hardware. This task is called *bundling.* After bundling, (full) versions for implementing the expression tree with root `source` exist. The result of the first call to `bundling` is shown in fig. 10. This call generates a version with root `SH.P1`.

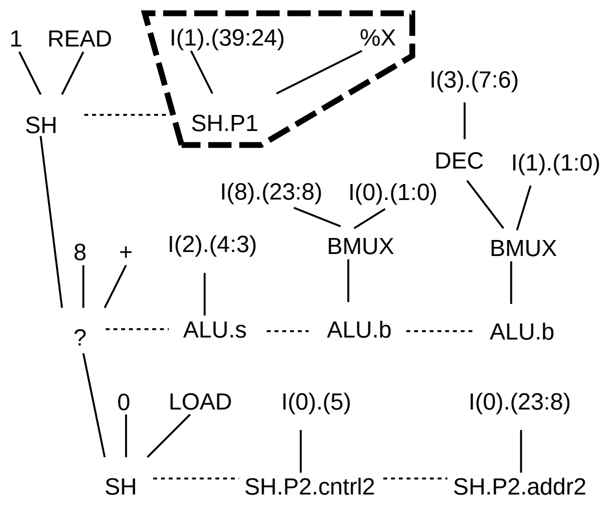

Figure 10: Result of bundling for node `SH`

The pseudo-code for bundling is the following:

```
PROCEDURE bundling(n : tree);
 BEGIN
  generate all possible (full) versions;
  delete partial versions linked to n;
 END;
```

A very important task of bundling is to detect resource conflicts: partial versions for the different input ports could be in conflict to each other. Both MSSV and MSSQ 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 conflict). 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 the main limitation of actual hardware is that no wire can carry two different signals at a time. Since multiplexers are explicit in our hardware model, avoiding conflicts at the control word is sufficient for guaranteeing conflict-free signal assignments. There is one exception, however: the *programmable bus conflict.* This case is handled separately (see below).

Bundling is slightly different for ordinary nodes and for catenation nodes (c. f. `catbundling`). Whereas `bundling` tries to find partial versions for all required ports, `catbundling` tries to find the partial versions for all required bits of one port.

After bundling, `expression` tries to find paths from the root of versions linked to `source` to components matching the `sink` node.

The call to `path` for `source = SH.P1` generates a partial version with root `ALU.a` (see fig. 11).

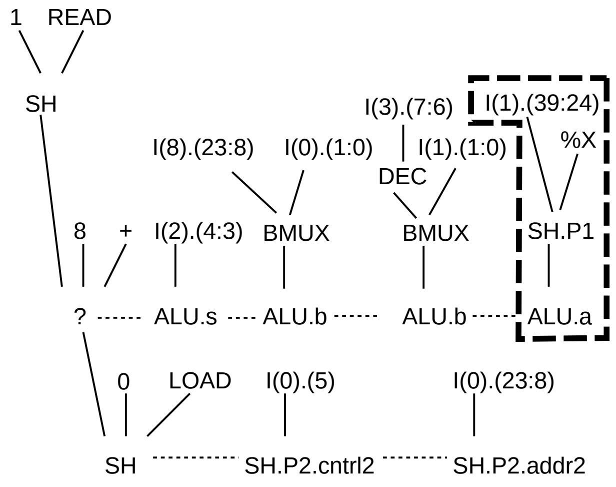

Figure 11: After calling `path` for `SH`

The outstanding completion of `expression` for the first two nodes results in additional calls to `bundling` and `path`. This leads to figs. 12, 13 and 14.

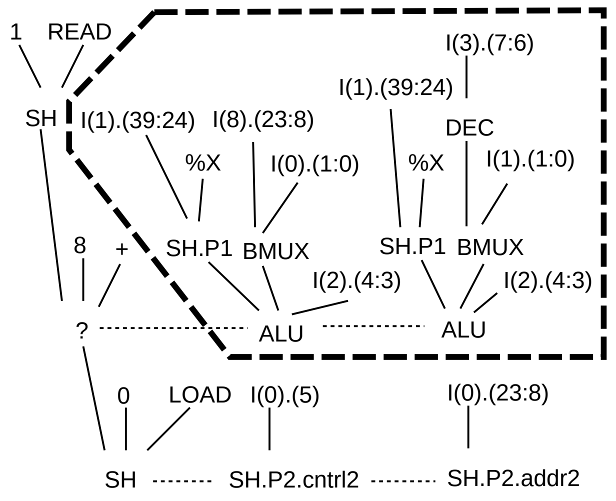

Figure 12: Result of bundling at ?

Note that the version containing `I(8).(31:16)` is not included in fig. 14 because it causes an instruction conflict with `I(0).(31:16)`.

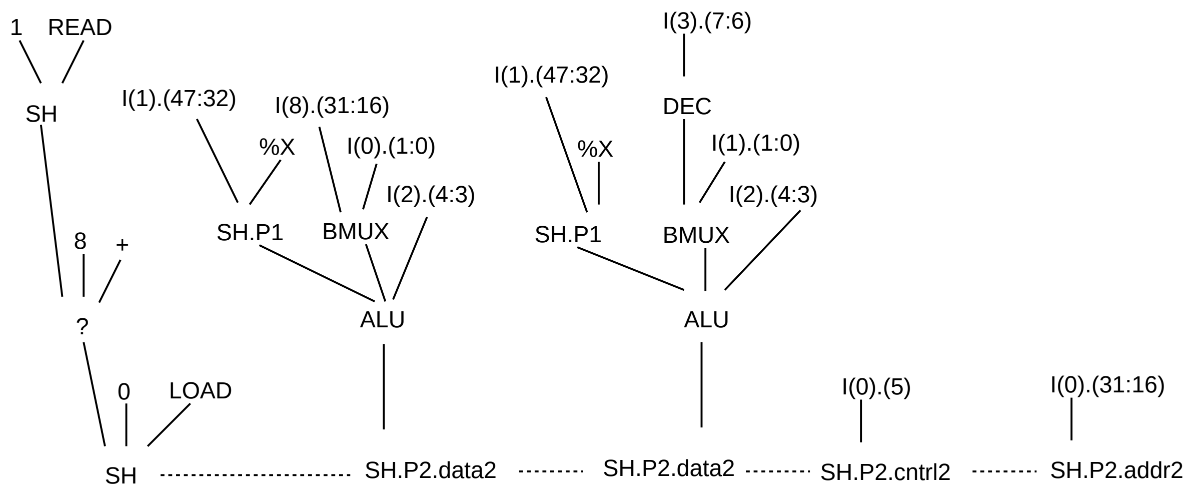

Figure 13: After calling `path` for ?

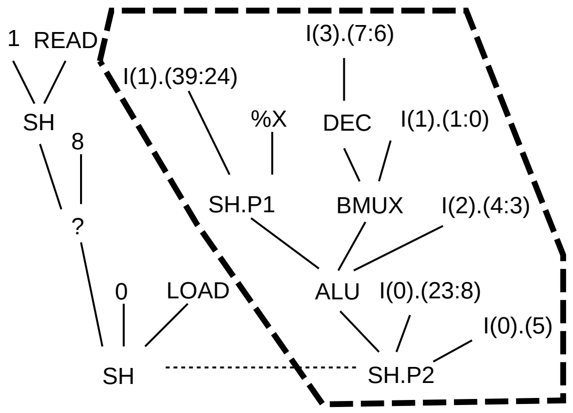

Figure 14: Result of bundling at 2nd `SH`-node

4.7 Extraction of binary code

The binary code can be extracted from fig. 14 tree quite easily (see table 1).

Bits	47:32	31:16	15:8	7:6	5	4:3	2	1:0
Code	1	0	X	3	0	2	X	2

Table 1: Binary code for the running example

5 Additional Tasks

5.1 Disabling buffers and storage devices

The code generated so far 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 ACCU (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.

A similar problem arises with busses. In some cases, bus drivers can be controlled such that multiple writes to a bus exist (*programmable bus conflict*). We avoid such conflicts by generating versions for the TRISTATE-operation of bus-drivers. The scheduling algorithm then guarantees that, for every control step, unused bus drivers are in TRISTATE-mode.

5.2 Scheduling

Generated version lists are fed into the scheduler. The scheduler is an ASAP-time scheduler with special provisions to handle alternate code sequences for each of the assignments. The scheduler tries to put as many statements into one control step as possible by checking available versions.

6 Limitations

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. Multi-cycle clocking is assumed to be invisible at the instruction level. Delayed jumps are not supported by MSSV. A linker has not yet been designed (although this would be straightforward).

Due to its limited speed, the mapper cannot be used for large algorithms (this is not a problem for DSP applications).

7 Conclusion

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.

The most important observation is that the number of versions for typical designs is significantly larger than expected. Hundrets of versions per statement are rather common for real designs. Unfortunately, this slows down the code generation considerably. Execution times of a few seconds per statement are no exception. Mappers for the applications at hand are not used for large algorithms, but the speed of MSSV would inhibit real applications. The reason for the large number of versions frequently originates from control code alternatives very 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 backtracking in case of conflicts does not help, because the number of backtracks is estimated to be in the order of the number versions which MSSV creates. And backtracking would cause much more overhead. Another problem of MSSV is that it cannot be used with large lists of components containing temporary locations.

We are therefore now using MSSQ instead of MSSV. MSSQ allows *version*-nodes (OR-lists) anywhere in the tree. This eliminates the cross-product problem. As a result, all code alternatives can be stored even in a small main memory. No alternative has to be ignored due to memory or run-time constraints and hence it can be guaranteed that MSSQ does not generate multiple control steps in cases where a solution with a single control step exists.

Although MSSQ(uick) is faster than MSSV, MSSV is ideal for demonstrating the basic principles in an emerging discipline. The key ideas of the matching process are easier to understand than for a complex system like MSSQ. This report on MSSV should contain enough information to stimulate further research in this emerging area.

References

[BH81] T. Baba and H. Hagiwara. The MPG system: A machine-independent microprogram generator. *IEEE Trans. on Computers, Vol. 30*, pages 373–395, 1981.

[BMSJ91] R. Beckmann, P. Marwedel, W. Schenk, and R. Jöhnk. The MIMOLA language reference manual - version 4.0 -. Technical Report 363, Computer Science Dpt., University of Dortmund, 1991.

[BPSJ91] R. Beckmann, D. Pusch, W. Schenk, and R. Jöhnk. The TREEMOLA language reference manual – version 4.0 –. Technical Report 391, Computer Science Dpt., University of Dortmund, 1991.

[EGO79] 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.

[Fis91] J. Fisher. RISC: The death of microcode, or the victory of microcode? *Proc. 24th Ann. Int. Symp. on Microarchitecture, keynote*, 1991.

[Fly90] M. Flynn. Instruction sets and their implementations. *Proc. 23rd Ann. Int. Symp. on Microarchitecture*, pages 1–6, 1990.

[GCLM92] G. Goossens, F. Catthoor, D. Lanneer, and H. De Man. Integration of signal processing systems on heterogeneous IC architectures. *6th ACM/IEEE High-Level Synthesis Workshop*, 1992.

[GFH82] M. Ganapathi, C.N. Fisher, and J.L. Hennessy. Retargetable compiler code generation. *ACM Computing Surveys, Vol. 14*, pages 573–593, 1982.

[IEE92] Design Automation Standards Subcommittee of the IEEE. Draft standard VHDL language reference manual. *IEEE Standards Department, 1992*, 1992.

[IFI93] IFIP. Workshop on hardware/software codesign. *Innsbruck, Austria*, 1993.

[LCG$^+$90] D. Lanneer, F. Catthoor, G. Goossens, M. Pauwels, J. Van Meerbergen, and H. De Man. Open-ended system for high-level synthesis of flexible signal processors. *1st EDAC, Glasgow*, 1990.

[Mar84] P. Marwedel. A retargetable compiler for a high-level microprogramming language. *ACM Sigmicro Newsletter, Vol. 15*, pages 267–274, 1984.

[Mav92] F. Mavaddat. Data-path synthesis as grammar inference. *IFIP-Workshop on Control Dominated Synthesis from A Register Transfer Description, Grenoble*, 1992.

[MN89] P. Marwedel and L. Nowak. Verification of hardware descriptions by retargetable code generation. *26th Design Automation Conference*, pages 441–447, 1989.

[MV83] R.A. Mueller and J. Varghese. Flow graph machine models in microcode synthesis. *17th Ann. Workshop on Microprogramming (MICRO-17)*, pages 159–167, 1983.

[Now87] L. Nowak. Graph based retargetable microcode compilation in the MIMOLA design system. *20th Annual Workshop on Microprogramming (MICRO-20)*, pages 126–132, 1987.

[TI91] Texas-Instruments. TMS320C2x. *User's Guide*, 1991.

[Veg82] S.R. Vegdahl. Local code generation and compaction in optimizing microcode compilers. *PhD thesis and report CMUCS-82-153, Carnegie-Mellon University, Pittsburgh*, 1982.

[War89] E. Warshawsy. Retargetable microcode generation from VHDL. *JRS Research Lab. Inc., Irvine, (personal communication)*, 1989.

[XIL91] XILINX. The XC 4000 data book. *San Jose*, 1991.