Graph Based Retargetable Microcode Compilation in the MIMOLA Design System

Lothar Nowak

Institut für Informatik und Praktische Mathematik, University of Kiel, Olshausenstr. 40-60, D-2300 Kiel, W. Germany

Abstract

This paper describes a retargetable compiler, which is able to compile programs into the machine code of a specified hardware (target). The target is described at register-transfer structure level by module specifications and netlists. The program can be defined at several levels of abstraction, spanning the range from algorithmic description (e.g. PASCAL) down to RT-level behavioural description. If the program is the complete target's behavioural specification the compiler can be used to verify the structural against this behavioural description.

1 Introduction

The general motivation for the design of retargetable compilers has been intensively discussed in the literature (see e.g. [Mar84]). It need not be repeated. Retargetable compilers require a formal description of the target machine, for which code is to be generated. Many different forms of target machine descriptions have been used. This paper describes a retargetable compiler using the true hardware structure as machine specification. This means that the specification consists of a description of the target's RT-modules and their interconnection. This approach has several advantages:

- easy integration into a general CAD-system,
- consistent machine description during the design process, useful for other tools, e.g. simulators,
- the specification language is easy to learn for hardware engineers.

This paper presents a retargetable compiler, which supports target architectures with single level interpretation and monophase execution. This class represents most of all microcoded systems; even multiphase systems can often be modelled by monophases. The input to the compiler is the struc-

Permission to copy without fee all or part of this material is granted provided that the copies are not made or distributed for direct commercial advantage, the ACM copyright notice and the title of the publication and its date appear, and notice is given that copying is by permission of the Association for Computing Machinery. To copy otherwise, or to republish, requires a fee and/or specific permission.

© 1987 ACM 089791-250-0/87/0012/0126 $1.50

Earlier work [Mar84] demonstrated that the approach outlined above is feasible. However, until recently, retargetable code generation was too slow to be used for large programs. The present paper describes a novel approach speeding up the compilation by a significant amount. The compilation process is divided into three phases: the preallocation phase, the allocation phase and the scheduling phase. The hardware allocation is based on the Connection-Operation-Graph (CO-Graph), constructed during the preallocation phase. This graph represents the target structure as well as all operation codes. In the allocation phase a pattern matching algorithm searches subgraphs, which are equivalent to the dataflow graph defined by the assignment to be allocated. The resource allocation is represented by the corresponding mapping of the instruction word. These settings are described by special structures, called I-Boxes. Finally, all allocated assignments are scheduled and mapped into instructions.

The main difference to other retargetable compilers can be seen in human interaction needed for the compilation. While, e.g. the system of Muehle et al. [Mue83, Mue84] relies on manually bound flowgraphs for data transforms, Veldahl's system [Veg82, Veg83, Veg88] uses procedural descriptions, e.g. how to generate constants. In contrast, our system's input is the pure structural description of the target. Another difference is given by the handling of versions: in general, one data transfer can be mapped to the target structure in different ways. This aspect was first stated by Maitlert [Mait78] but not considered in retargetable code generation. Versions offer more freedom in the scheduling phase [Mue84] and are an essential concept of our compiler.
Def. 2.1:
Let $B_2 := \{0, L\}$ and $B^n_2 := P(B^n_2)$ the power set of $B_2$.

The elements of $B^n_2$ are:

$\emptyset := \{\}, \emptyset := \{0\}, \{L\}, X := \{0, L\}$

With binary operations '+', '*' and a unary operation '-' as stated below ($B^n_2$, '+', '*', ') define a boolean algebra.

<table>
<thead>
<tr>
<th></th>
<th>0 0 1 X</th>
<th></th>
<th>0 0 0 0</th>
<th>a</th>
<th>-a</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\emptyset$</td>
<td>0 0 1 X</td>
<td>$\emptyset$</td>
<td>$\emptyset \emptyset \emptyset \emptyset$</td>
<td>$\emptyset$</td>
<td>$X$</td>
</tr>
<tr>
<td>0</td>
<td>0 0 0 0</td>
<td>0</td>
<td>$\emptyset \emptyset \emptyset \emptyset$</td>
<td>0 1</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>$1 X X X$</td>
<td>1</td>
<td>$\emptyset \emptyset 1 1$</td>
<td>1 0</td>
<td></td>
</tr>
<tr>
<td>X</td>
<td>$X X X X$</td>
<td>X</td>
<td>$\emptyset 0 1 X$</td>
<td>X</td>
<td></td>
</tr>
</tbody>
</table>

An interpretation of this algebra is given by the following treatment. Assume a (boolean) variable $A$ (e.g. a bit of the instruction word) and some conditions, which assert $A$ to be set to specific values. Distinguish four cases:

- $A$ should be set to $\{0\}$, write $A:=0$
- $A$ should be set to $\{L\}$, write $A:=1$
- $A$ should be set either to $\{0\}$ or to $\{L\}$, write $A:=01$
- $A$ could never be set to $\{0\}$ and $\{L\}$ simultaneously, write $A:=01$

The value $'X'$ ($\{0, L\}$) offers a choice between the alternatives $'0'$ or $'1'$. The value $'\emptyset'$ ($\{\}$) can be treated as the result of some incompatible assertions. Since $(B^n_2, +, *, \neg)$ is a boolean algebra the distributive law holds; for example: $1*(1+0) = 1*X = 1 = 1*$ and $(1*) = (1*) (1*)$.

To handle $n$-dimensional variables (e.g. the complete instruction word) an extended algebra using boolean vectors can be defined.

Def. 2.2:
Let $B^n_2 := \{(b_1, b_2, \ldots, b_n) | b_i \in B_2\}$ and $B^n_2 = P(B^n_2)$ the power set of $B^n_2$.

$(B^n_2, +, *, \neg)$ with '+', '*' and '-' defined by the set-theoretic operations union, intersection and complement defines a boolean algebra.

Equivalent to the 1-dimensional case $B_2 = B^n_2$, an element of $B^n_2$ (e.g. the set $(v_1, v_2, v_3)$, $v_i \in B_2$) can be interpreted as the alternative settings of $(v_1, v_2)$ or $v_3$ of a $n$-dimensional boolean variable. The interpretations of union and intersection are obvious. Since $B^n_2$ contains $2^n$ elements, a compact representation of $B^n_2$ must be found: it is given by I-Trees, a recursively defined structure composed of I-Nodes.

Def. 2.3:
Let $B^n_2 := \{(b_1, b_2, \ldots, b_n) | b_i \in B_2, \emptyset, \{0, 1\}, \{1\}, X \}$

The n-dimensional vector $v \in B^n_2$ is called I-Node

Let $v = (v_1, v_2, \ldots, v_n)$ I-Nodes. Define the operation $\times_1$ : I-Node x I-Node -> I-Node by

$v_1 \times_1 v_2 = (v_1 \times_1 v_2, \ldots, v_n \times_1 v_2),$

I-Nodes can be mapped into elements of $B^n_2$ by a mapping $F$:

Def. 2.4:
Let $v = (v_1, v_2, \ldots, v_n) \in B^n_2$

$F : B^n_2 \rightarrow B^n_2$ with

$v_1 \times_1 v_2 \Rightarrow F(v) = \emptyset$

$v_1 \times_1 \emptyset \Rightarrow F(v) = (v_1, \ldots, v_{i-1}, \emptyset, \ldots, v_n),$

$v_1 \times_1 v_j \Rightarrow F(v) = (v_1, \ldots, v_{i-1}, v_i = \emptyset, \ldots, v_j) \times_1 v_j, v_j \times_1 v_i \Rightarrow F(v) = (v_1, \ldots, v_{i-1}, v_i \times_1 v_j, \ldots, v_j) \times_1 v_i$

It can be shown that $F(a \times_1 b) = F(a) \times F(b)$ holds if $a,b$ are I-Nodes. Due to $F$'s property $F(B^n_2) \subset B^n_2$ an inverse mapping cannot be defined. But let's have a look at $B^n_2$. If $v_1$ denotes an element of $B^n_2$, the elements of $B^n_2$ are (1) $v_i$ or sets of the form $\{v_1, \ldots, v_j\}$, the latter one is equivalent to the boolean expression $v_1 * + \ldots v_j$. Therefore, each element of $B^n_2$ can be expressed by an I-Node (e.g. by $\emptyset \emptyset \emptyset \emptyset \emptyset$ or $v_1$) or by a boolean expression of I-Nodes (e.g. $v_1, \ldots, v_j$) by $v_1 * \ldots v_j$. Having this in mind we define a structure representing algebraic expressions of I-Nodes.

Def. 2.5:
An I-Tree is a recursively defined structure composed of I-Nodes and the two relations '+', and '*'. It is an:

1) I-Node
2) I-Node * I-Tree
3) I-Tree + I-Tree

An I-Tree can be graphically represented by a tree, using following conventions ($A, B \in I$-Tree):

- $A + B \Rightarrow B$ is brother of $A$
- $A \times B \Rightarrow B$ is son of $A$
- (i) demands for a dummy root node, e.g.:

For example:

The algebraic expression $A * (B + C + (D * E)) + F$ and its distributive form $A * B + A * C + \ldots + F$ is represented by the following left (right) I-Tree:
Since I-Trees are representations of algebraic expressions commutativity, associativity and distributivity holds on I-Trees too. Using distributive law each I-Tree can be transformed into a disjunctive form like

\[ \sum_{i=1}^{k} \prod_{j=1}^{n} v_{ij} \cdot v_{ij} \text{ I-Nodes} \]

Due to the correspondence between the operations '+' and '*' I-Trees can be reduced to expressions of the form

\[ \sum_{i=1}^{k} v_{i}, \cdot v_{i} \text{ I-Node} \]

Therefore, each I-Tree represents an element of \( E_{2}^{n} \) \( E_{2}^{n} \)'s operations ('+', '*') correspond to representations on I-Trees. Therefore, these operations could be done on I-Trees directly. While operation '+' is already defined on I-Trees the operation '*' must be extended from I-Node x I-Tree to I-Tree x I-Tree. This can be done using distributive law. To avoid some confusion the corresponding operations are called 'merge' and 'cut', respectively (let \( A, B, R \in \text{I-Tree} \)).

Def. 2.6:

merge : I-Tree x I-Tree \( \rightarrow \) I-Tree,
merge(\( A, B \)) := \( A + B \)

Def. 2.7:

cut : I-Tree x I-Tree \( \rightarrow \) I-Tree,
cut(\( A, B \)) := \( R \)

with \( R := \text{FOR all leaves } a_{i} \text{ of } A \)
\( \text{DO } a_{i} := a_{i} \cdot B \) (distributive law)

The example, given above, demonstrated that all conjunctive terms can be found as pathes from the root to a leave without regard of the representation. A path \( P \) is called conflicting, iff \( \forall a, b \in P, a \cdot b \neq (\ldots, \emptyset, \ldots) \). Since \( (\ldots, \emptyset, \ldots) \) is equivalent to the void set \( \{\} \) all conflicting pathes can be removed from I-Trees. This cleanup should be made after each cut operation to keep the resulting I-Trees small.

3 Definition of the Connection-Operation-Graph

The Connection-Operation-Graph (CO-Graph) is the key structure used in the allocation phase. It represents the target structure as well as all operations the target is able to perform and their operation codes. The target's description level is the register-transfer structure level. It consists of the behavioural specifications of RT-modules and a netlist, describing the interconnection of modules.

The specification of a RT-module contains information about
- the interface, e.g. inputs, outputs and their widths,
- the operations performed by the module and their corresponding control-codes and
- the timing, e.g. clock, delay-times.

Fig. 3.1 shows an example of a RT-module described in MIMOLA ( '".' denotes a binary constant, the delay-times are measured in time units).

\[ \text{MODULE Alu (IN in1,in2; \text{word);} \}
\]
\[ \text{BEGIN} \]
\[ \text{CASE \text{ctr OF}} \]
\[ \text{\$00 : \text{outp} \leftarrow \text{in1} + \text{in2} \text{AFTER 10;}} \]
\[ \text{\$01 : \text{outp} \leftarrow \text{in2} - \text{in1} \text{AFTER 10;}} \]
\[ \text{\$10 : \text{outp} \leftarrow \text{in1} \text{AFTER 5;}} \]
\[ \text{END} \]
\[ \text{END;}} \]

Fig. 3.1: Example of a MIMOLA RT-module description

Those RT-level module descriptions can be expressed by M-Graphs. A M-Graph is a directed acyclic graph, representing the module specification. It consists of a list of operation trees linked at a common root node, equivalent to the module output. Each tree represents one operation of the module and the argument inputs. Special M-Graphs are used for hardwired constants and for fields of the instruction word. An example of a M-Graph is given in fig. 3.2. Some additional information, e.g. widths and timing, is attribute of the nodes and not shown.

Fig. 3.2: Example of a M-Graph (see fig. 3.1)

Due to the underlying model of the synchronous automaton, loadable modules (registers, memories) can be used twice per state transition: they can be read and loaded. This is reflected in our treatment by splitting registers and memories into two M-Graphs. One graph represents all load-operations, the other one all read-operations. The ports of multiport-memories are treated as separate modules.

M-Graphs are used to construct the Connection-Operation-Graph (CO-Graph). The CO-Graph is a directed
structured using the MIMOLA hardware description of the target. Additionally, some local transformations are done; e.g., insertion of commutated or converted operation trees, generation of vias (see below) and transformations handling busses. The operation codes are now treated as assertions for the corresponding control input: the operation tree input must be supplied with the code value (or alternative values). Operations may correspond to one, more or even no assertion. Additionally, assertions are produced by the generation of vias. A via is defined by an operation, which does not modify data if the remaining argument(s) is set to a specific value (the neutral element). An example is given by 'a + 0' or by 'a AND #FF'. Fig. 4.1 shows a part of the modified CO-Graph, commutated operations are not shown. Assertions are denoted by ';', the operation 'dat' is equivalent to the identity operation.

![Fig. 4.1: Alu's M-Graph with assertions](image)

Next it is tried to satisfy all assertions found at the CO-Graph. Since assertions are constants to be delivered at certain module inputs, this can be done by programming the target. The corresponding instruction codes are represented by I-Trees and attached to assertion's operation trees. If more than one assertion is attached to an operation the resulting I-Tree is the out of all corresponding I-Trees. The I-Trees can be generated by the following algorithm.

```plaintext
PROCEDURE alloc_const(value,m_graph,i_tree) {1}
  BEGIN
    i_tree:={};
    CASE m_graph OF
      hard_const:
        IF Const_value=value
        THEN i_tree:={(X,X,...,X)};
      inst_field:
        i_tree:={(X,X,...,value...)};
      OTHERWISE:
        FOR all operations of m_graph DO {4}
          IF operation='dat/via' THEN
            BEGIN
              alloc_const(value, arg(operation), tmp);
              tmp:=Out(tmp, operation_i_tree);
              IF tmp<>{} THEN i_tree:=merge(i_tree, tmp);
            END
        END_CASE
      END;
  END;
```

In the preallocation phase the CO-Graph is connected by directed arcs according to the interconnection structure given by the netlist. Additionally, all data sinks (registers and memories) are linked at a common node: the CO-Graph root. Via this node all M-Graphs can be accessed. An example of a target and its CO-Graph is shown in fig. 3.3, and fig. 3.4, respectively.

![Fig. 3.3: Example of a target](image)

![Fig. 3.4: Equivalent CO-Graph](image)

(sequels of M-Graphs not shown)
The algorithm must be called with the assertion value and the first M-Graph linked at the assertion node (arg(assert_node)). The result is returned at i_tree.

The M-Graph represents a hardwired constant: if it matches the value is found and a dummy I-Tree generated, otherwise the I-Tree is empty.

The M-Graph represents a field of the instruction word: the field is set to the value.

The M-Graph represents a modular: all operation trees are tested.

A dat-operation is found: try to get the value recursively.

To route the value found, the correct operation must be selected: this is guaranteed by operation's I-Tree.

One version is found: merge it with previously found ones.

The algorithm assumes that the I-Trees of all reachable dat-operations are already known. Therefore, constant allocation has to be done in a certain order. Care must be taken in case of cyclic dependencies, caused by such unknown I-Trees.

The I-Trees found during the allocation are linked at their operation nodes. For all satisfied assertions the corresponding inputs are deleted. Operations with unsatisfied assertions can never be executed and are deleted too. The result is the preallocated CO-Graph. An example is shown in Fig. 4.2, the I-Trees are represented by [...]. The following instruction bit assignment is assumed: I.R0, I.R1, I.Mux: (...3), I.Alu: (2:1), I.Inm: (0)

Fig. 4.2: Preallocated graph (only Alu shown)

5 The Allocation Phase

This part performs the statement allocation. It is based on a pattern matching algorithm. This algorithm tries to find all subgraphs of the preallocated CO-Graph, which are equivalent to the dataflow graph defined by the assignment to be allocated. Equivalent means: the structures must be equal, but additional dat/via-operations are allowed. Additionally, all I-Trees found at the operation nodes must be compatible. If an equivalent structure with compatible I-Trees is found, the out of these trees represents all versions for (1-cycle) execution of the assignment. It is stated that no distinction is made between data and control effects of statements: in this sense a conditional jump is an assignment to the program counter register using a special conditional-operation. A rough description of the allocation algorithm is given below.

PROCEDURE alloc_assign(a_node,m_graph,i_tree)
BEGIN
  i_tree := ();
  FOR all operations of m_graph DO
    IF operation = a_node THEN
      BEGIN
        tmp1 := (all);
        FOR all arguments of operation/a_node DO
          BEGIN
            alloc_assign
              (arg(a_node), arg(operation), tmp2);
            tmp1 := out(tmp1, tmp2);
          END;
        tmp1 := out(tmp1, operation_itree);
        IF tmp1 <> () THEN
          i_tree := merge(i_tree, tmp1);
        END;
      ELSE
        IF operation = 'dat/via' THEN
          BEGIN
            alloc_assign(a_node, arg(operation), tmp1);
            tmp1 := out(tmp1, operation_itree);
            IF tmp1 <> () THEN
              i_tree := merge(i_tree, tmp1);
            END;
          END;
      END;
  END;
END;

Notes:
1) The algorithm is called with the assignment root and the matching M-Graph (sink node of the corresponding storage location). The result is returned at i_tree.
2) Test all versions of the M-Graph for matching operations and for dat/via-operations.
3) A matching operation is found: test all arguments recursively.
4) Cut the resulting argument and operation I-Trees.
5) One version is found: merge it with previously found ones.
6) A dat/via-operation is found: try to get a matching operation recursively.

If an assignment cannot be allocated, a detailed error report, (e.g., "no path from ... to ...", "instruction conflict at ...") is generated by alloc_assign. This error report is then used for selecting a subexpression which will be assigned to a temporary variable. The original assignment will be split into a sequence of two statements. Allocation continues for both parts recursively. Special user defined storage locations (memory cells, registers) are used as temporary variables.

6 The Scheduling Phase

The input to this phase is a collection of allocated assignments, which originate from a parallel block (or straight lines of those blocks). The resource allocation is represented by I-Trees. A schedule must be found, which maps the assignments into instructions. The assignments are partially ordered by the two relations 'data dependent' and 'data anti-dependent' (a formal definition can be found in [Mar85]).
Def. 6.1:
Let \( S_1, S_2 \) be assignments.

\( S_2 \) is data dependent on \( S_1 \), iff \( S_1 \) writes into a storage location, which will be read by \( S_2 \).

\( S_2 \) is data anti-dependent on \( S_1 \), iff \( S_2 \) 'overwrites' a storage location still to be read by \( S_1 \).

For example:
\[ S_2 \text{ data dependent on } S_1 : \]
\[ \begin{array}{l}
\text{SEQBEGIN } R1 := \ldots ; \ldots := R1 \text{ SEQEND.} \\
\text{FARBEGIN } ... := R1 \otimes R1 := \ldots \text{ PAREND.}
\end{array} \]

Def. 6.2:
Let \( S_1, S_j \) be assignments.

\[ \text{GE}(S_1) := \{ S_j \mid S_j \text{ data anti-dependent on } S_1 \} \]

\[ \text{GT}(S_1) := \{ S_j \mid S_j \text{ data dependent on } S_1 \} \]

\( S_j \in \text{GE}(S_1) \) is denoted by \( S_1 \leq S_j \).

\( S_j \in \text{GT}(S_1) \) is denoted by \( S_1 < S_j \).

The greater-than and the greater-equal relations can be interpreted in terms of schedule times. The corresponding sets are constructed for all assignments of the input collection.

Def. 6.3:
A collection \( C \) of assignments is called cyclic dependent, iff \( S_1 \leq S_2 \leq \ldots \leq S_n \leq S_1 \) and \( S_1 \in C \).

A cyclic dependency can only be satisfied by \( S_1 = S_2 = \ldots = S_n \). Therefore, all cycle members must be scheduled at one instruction. This can only be done if the resulting I-Tree \( I_{yc} = \text{cut}(I_1, \ldots , I_n) \) is not empty. A cycle counter is represented by \( I_{yc} \) and treated like a single assignment. GE- and GT-sets must be updated accordingly.

Each instruction contains a set of registers and memories to be loaded and implicitly the complementary set of storages to be not loaded. The latter one is denoted by the noop set. To avoid unintended state transitions all members of this set must 'execute' the no-load-operation (disable writing). The corresponding instruction codes are represented by I-Trees too. Additionally, all versions to increment the program counter (including, e.g. the jump at the following instruction) are treated as a special noop. It is added to the set, if no explicit jump is present. If \( N_1, \ldots , N_n \) are the I-Trees of the noop set under consideration, then \( \text{Inoop} = \text{cut}(N_1, \ldots , N_n) \) is a precondition for all instructions to be scheduled currently.

If \( \text{GT}(S) = \{ \} \) and \( \text{GE}(S) = \{ \} \) holds for an assignment \( S \), \( S \) is ready (to be packed). The definition of 'ready' shows a main objective of the scheduling algorithm: the last instruction is packed first. Due to the np-completeness of the problem, the algorithm uses heuristics.

PROCEDURE schedule(\( S_1, \ldots , S_n, \text{inst_list} \))
BEGIN
  generate GE- and GT-sets for all \( S_i \);
  handle cyclic dependencies;
  \[ \text{REPEAT} \]
  \[ \text{generate common noop set} \]
  \[ \text{for all } S_i \text{ with } \text{GT}(S_i) = \{ \} ; \]
  \[ \text{i_tree} := \text{cut(all I-Trees of noop set)} ; \]
  \[ \text{REPEAT} \]
  \[ \text{Sx} := \text{get one of } S_i, \text{ which is:} \]
  \[ \text{ready, not packed, not tested}; \]
  \[ \text{IF cut(i_tree, Sx_itree) <> \{ \}} \text{ THEN} \]
  \[ \text{i_tree} := \text{cut(i_tree, Sx_itree)} ; \]
  \[ \text{remove Sx from all GE-sets}; \]
  \[ \text{END} \]
  \[ \text{UNTIL all tested}; \]
  \[ \text{cut i_tree with remaining noops}; \]
  \[ \text{set unused busses to tristate}; \]
  \[ \text{link i_tree at inst_list}; \]
  \[ \text{remove all packed SI from GT-set}; \]
  \[ \text{reset tested flags}; \]
  \[ \text{UNTIL all packed}; \]
  \[ \text{END} \]

Notes:
1) The algorithm gets a collection of assignments and returns a sequence of instruction versions, represented by I-Trees.
2) Only assignments with empty GT-set are candidates for the next instruction to be packed.
3) An instruction will be generated and is set up with the noop's I-Trees.
4) All assignments in the ready state are tested for compatibility.
5) A compatible assignment is found: it is added to the instruction; satisfied data anti-dependencies can be removed.
6) If all assignments are tested, the instruction is closed.
7) Some noops may be in the complement of the instruction's load set but not in the noop set. Multiport memories need a special handling.
8) Bus usage is a sideeffect and must be handled similar to the noops.
9) The preceding instruction is generated next: corresponding data dependencies are satisfied.

The example shown in fig.6.1 is based on a true hardware design (a processor constructed of AMD 29203 bit slices). Allocation and scheduling are hand capped by the use of memories with common read/write addresses. The order of scheduling depends on heuristics: if several assignments are ready, the first one is chosen (SR[2] denotes a cell with address 2 of the register bank SR, RQ is a register, [x] denotes symbolic addresses).

7 Results
In this paper a retargetable compiler suited for a large class of targets is presented. In contrast to other known compilers retargetability is achieved by a pure structural description of the target. No additional information is needed. Because an error message is returned if an assignment cannot be allocated, the compiler can be used as a verifier: the behavioral target description
Assignments at the register-transfer level:

```
PARBEGIN
((SR[3]*SR[4])+SR[2]);
PAREND;
```

Sequentialization (temp.location: SR[1...5]):

1st assignment:
```
BEGIN
  1) SR[41] := SR[1];
  2) SR[41] := SR[41]+SR[3];
  3) SR[2] := SR[41];
END;
```

2nd assignment:
```
BEGIN
  4) SR[42] := SR[3];
  6) SR[43] := RQ*SR[2];
  8) SR[44] := SR[3];
 13) SR[1] := SR[45];
END;
```

List of the assignments, one order of packing:

<table>
<thead>
<tr>
<th>Stat</th>
<th>GE</th>
<th>GT</th>
<th>order</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>13</td>
<td>2</td>
<td>4</td>
</tr>
<tr>
<td>2</td>
<td>-</td>
<td>3</td>
<td>2</td>
</tr>
<tr>
<td>3</td>
<td>-</td>
<td>-</td>
<td>1</td>
</tr>
<tr>
<td>4</td>
<td>-</td>
<td>5</td>
<td>12</td>
</tr>
<tr>
<td>5</td>
<td>3</td>
<td>7</td>
<td>11</td>
</tr>
<tr>
<td>6</td>
<td>3</td>
<td>7</td>
<td>13</td>
</tr>
<tr>
<td>7</td>
<td>-</td>
<td>11</td>
<td>10</td>
</tr>
<tr>
<td>8</td>
<td>-</td>
<td>9</td>
<td>8</td>
</tr>
<tr>
<td>9</td>
<td>-</td>
<td>10</td>
<td>7</td>
</tr>
<tr>
<td>10</td>
<td>3</td>
<td>12</td>
<td>6</td>
</tr>
<tr>
<td>11</td>
<td>-</td>
<td>12</td>
<td>9</td>
</tr>
<tr>
<td>12</td>
<td>-</td>
<td>13</td>
<td>5</td>
</tr>
<tr>
<td>13</td>
<td>-</td>
<td>-</td>
<td>3</td>
</tr>
</tbody>
</table>

Fig. 6.1: Example for scheduling (AMD 29203)

has to be compiled using its structural description.

The resource allocation is mapped to the handling of I-Trees: an algebraic representation of instruction word versions and their relations. This can be done because resource conflicts lead (with the exception of buses) to instruction code conflicts. Using I-Trees, all data transfer versions can be handled at once. Even in the scheduling phase the I-Tree representation is used. This results in a scheduling algorithm, which considers versions of allocated assignments and tests them all for compatibility. Versions should not be ignored: in some cases we found more than 100 different versions for the execution of an assignment. Additionally, the scheduling algorithm handles instruction-operations (disable writing). Explicitly, classical scheduling theories consider a default setting of the instruction word and ignore the problem. In reality this can be done only for a limited class of processors: e.g. this does not hold for all processors using the AMD 2910 sequencer.

The compiler is implemented as one tool of the MIMOLA Design System. Machine code was generated for many designs. Compilation rates about 10 to 1 generated instructions per second are achieved. This even holds for complex designs. For example, the CO-Graph of such a design [Kle86] consists of 66 K-Graphs with 233 operation trees.

8 References


