EINDHOVEN UNIVERSITY OF TECHNOLOGY
DEPARTMENT OF ELECTRICAL ENGINEERING
DESIGN AUTOMATION GROUP

HARDWARE ALLOCATION
BASED ON DIVIDE AND CONQUER TECHNIQUES

W.G.J. Volleberg

Master thesis
reporting on graduation work
performed from 15.03.87 to 15.01.88
by order of prof. dr.-ing. J.A.G. Jess
and supervised by ir. L. Stok

The Eindhoven University of Technology is not responsible
for the contents of training and thesis reports.
ABSTRACT

Due to the ever increasing complexity of today's VLSI designs costs are growing rapidly. Therefore the use of silicon compilers becomes necessary. A silicon compiler is a system that automatically transforms a functional description into a layout description of a design. Hardware structure synthesis is the highest part of a silicon compiler and translates a data flow graph description into a description of a data path and a controller.

In the silicon compiler currently under development the hardware synthesis is based on dynamic programming techniques resulting in rather large computation times and huge memory requirements. An alternative approach is based on divide and conquer techniques. Therefore the synthesis is divided into two mayor parts. The first part consists of assignment of demand graph nodes into machine cycles. The second part takes care of the demand graph nodes allocation. In contrast to the dynamic programming, the two parts are done separately.

In this report the demand graph node allocation is described.

The allocation task is divided into three allocations. They are variable allocation, operation allocation and interconnection allocation. Variable allocation maps variables onto registers and nets. Selecting of a minimum number of registers is implemented as a minimum colouring of nodes of an interval graph resulting in fast computation times. Operation allocation maps demand graph nodes onto library modules. Interconnection allocation maps interconnections between demand graph nodes onto nets and multiplexers. Operation allocation and interconnection allocation are implemented as a clique covering of a weighted graph. Heuristics are used to generate a near optimum solution in a reasonable amount of time.

After these allocations a data path is generated. Also the incomplete symbolic controller description, generated during the scheduling, is extended resulting in a complete symbolic description of the controller.

Hardware allocation based on the above techniques are less time and memory consuming than synthesis based on dynamic programming. The results of both methods are almost identical. The results are encouraging for further research.

The algorithms presented in this report are written in HotLisp, a dialect of CommonLisp.
PREFACE

This report is a result of my work done during my graduated period in the Automatic System Design Group (ES) of the faculty of Electrical Engineering at the Eindhoven University of Technology. In this group research and development is done on projects concerning CAD-tools and VLSI designs.

One of these projects, called Esprit-991 and supported from the EEG, deals with Silicon Compilation. Silicon compilation is an automatic generation of a circuit layout description from a behavioural (algorithmic) description. Due to the ever increasing complexity of todays VLSI designs the design costs are growing rapidly. The use of silicon compilers reduces the design costs and develop time. In the last decade several attempts on silicon compilers have been published.

In chapter 1 an overview of the silicon compiler is given. In chapter 2 the output of the scheduler and the pre processor are defined. In chapter 3 the variable allocation is given. Chapter 4 deals with the operation allocation and the details of the interconnection allocation are given in chapter 5. The extensions made whenever the variable allocation and operation allocation are reversed is given in chapter 6. In chapter 7 the generation of the final data path and the generation of the symbolic controller will be treated. Finally in chapter 8 some words about the implementation are spent.

At this place I would like to thank Automatic Design Group (ES) of the faculty of Electrical Engineering at the Eindhoven University of Technology for the support given. Especially I would like to thank prof. J.A.G. Jess, who made this research project possible, and L. Stok for the continuous support during my research and the proof reading of this report.

Wilgert Volleberg
6.2.3 Variable allocation versus extended variable allocation. 44

7. POST PROCESSING THE ALLOCATION RESULTS. 46
   7.1 STATE MACHINE EXTRACTION. 46
      7.1.1 Description of the state machine before allocation. 46
      7.1.2 Description state controller after allocation. 48
   7.2 DATA PATH EXTRACTION. 48
      7.2.1 Syntax of module list. 49
      7.2.2 Syntax of net list. 49

8. CONCLUSIONS. 50

REFERENCES 51

APPENDIX 0: Some words about the implementation.
APPENDIX 1: Overview *time-slots* syntax.
APPENDIX 2: Overview constant-list, port-list and array-list syntax.
APPENDIX 3: Overview *group-list* syntax.
APPENDIX 4: Overview *operator-vector* syntax.
APPENDIX 5: Overview *clique-graph* syntax.
APPENDIX 6: Overview *register-bank-vector* syntax.
APPENDIX 7: Overview extension of variable allocation syntax.
APPENDIX 8: Overview module-list syntax.
APPENDIX 9: Overview state-machine syntax.
# LIST OF FIGURES

<table>
<thead>
<tr>
<th>Figure</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.1</td>
<td>Overview silicon compiler.</td>
<td>2</td>
</tr>
<tr>
<td>1.2</td>
<td>Example of demand graph.</td>
<td>3</td>
</tr>
<tr>
<td>1.3</td>
<td>Overview of two techniques for hardware generation.</td>
<td>3</td>
</tr>
<tr>
<td>1.4</td>
<td>Different order of operation and variable allocation.</td>
<td>6</td>
</tr>
<tr>
<td>2.1</td>
<td>Pictorial example of <em>time-slots</em>.</td>
<td>8</td>
</tr>
<tr>
<td>3.1</td>
<td>Example of intersecting intervals.</td>
<td>14</td>
</tr>
<tr>
<td>3.2</td>
<td>Resulting variable interval graph of example.</td>
<td>14</td>
</tr>
<tr>
<td>3.3</td>
<td>Situation of invalid condition 2.</td>
<td>16</td>
</tr>
<tr>
<td>3.4</td>
<td>No variable interval graph due to loops.</td>
<td>18</td>
</tr>
<tr>
<td>3.5</td>
<td>Variable interval graph can be constructed.</td>
<td>18</td>
</tr>
<tr>
<td>4.1</td>
<td>Example of increase of delay.</td>
<td>24</td>
</tr>
<tr>
<td>4.2</td>
<td>Cost according to Tseng.</td>
<td>26</td>
</tr>
<tr>
<td>4.3</td>
<td>Example of data operation graph $DOG$ and sub operation graph $SOG$.</td>
<td>27</td>
</tr>
<tr>
<td>4.4</td>
<td>Optimisation of the in sets.</td>
<td>29</td>
</tr>
<tr>
<td>5.1</td>
<td>Example of a register file.</td>
<td>30</td>
</tr>
<tr>
<td>5.2</td>
<td>Non common sources or destinations can lead to increase of area.</td>
<td>31</td>
</tr>
<tr>
<td>5.3</td>
<td>Example of register forbid graph and register prefer graph.</td>
<td>33</td>
</tr>
<tr>
<td>5.4</td>
<td>Register prefer graph is not a chordal graph.</td>
<td>34</td>
</tr>
<tr>
<td>5.5</td>
<td><em>time-slots</em> description of example.</td>
<td>36</td>
</tr>
<tr>
<td>5.6</td>
<td>Resulting implementation without use of register files.</td>
<td>37</td>
</tr>
<tr>
<td>5.7</td>
<td>Final implementation after net optimisation.</td>
<td>38</td>
</tr>
<tr>
<td>6.1</td>
<td>New multiplex cost estimation.</td>
<td>40</td>
</tr>
<tr>
<td>6.2</td>
<td>The six connection sets CS of example.</td>
<td>41</td>
</tr>
<tr>
<td>6.3</td>
<td>The VPG graph of the example.</td>
<td>42</td>
</tr>
<tr>
<td>7.1</td>
<td>Overview of control nets.</td>
<td>47</td>
</tr>
</tbody>
</table>
LIST OF TABLES

TABLE 4.1. Example of implemented operations in the library. . . . . . . 21
TABLE 6.1. Life time analysis of example . . . . . . . . . . . . . . . . . 42
1. INTRODUCTION.

The highest level of a silicon compiler is often called hardware structure synthesis. Hardware structure synthesis schedules and allocates the demand graph nodes. Currently a method based on dynamic programming is used for the hardware structure synthesis resulting in rather large computation time and huge memory requirement. The main characteristic of dynamic programming is that scheduling and allocation are done concurrently. An alternative method is based on divide and conquer techniques. The scheduling and the allocation is done separately. In this report the allocation of demand graph nodes is described.

In paragraph 1.1 an overview of the mayor silicon compiler tools is given. In paragraph 1.2 special attention is paid to the hardware generator. A more elaborate definition of the silicon compiler can be found in [Stok86] or in [Jess87a].

1.1 SYSTEM ARCHITECTURE.

In figure 1.1 a global scheme of the silicon compiler is given. The system is partitioned into data description formats and tools. Ellipses represent tools; data description formats are represented as boxes. The task of a tool is to convert one data description format into another. Three description levels can be distinguished, i.e. an algorithmic level, a register-transfer level and a layout level. The tools used in a specific level are further declared.

The input of the algorithmic level consists of a behavioral description of the function fulfilled by the chip. The description is specified in a high level language. Besides the high level description constraints can be given. Constraints could be the power consumption, available area, pin configurations, timing, used technology etc. It is clear that changing the constraints leads to different results. The parser syntactically analyses the behavioral description and converts it into an abstract syntax tree. The next step is the translation of the abstract syntax tree into a demand graph.

The demand graph represents both data flow and control flow of a system. Nodes in the demand graph represent operations on data; arcs represent the direction in which data flows. The demand graph optimiser converts the demand graph into a functionally equivalent demand graph using 'standard' compiler techniques resulting in a more efficient demand graph description. An example of a demand graph is given in figure 1.2. The example stems from [Tseng86]. In figure 1.2 the variables $v_1$, $v_2$, $v_4$, $v_6$ and $v_{10}$ are externally provided to the system through an input pin. An input pin is represented as a get node in the demand graph. The results of the computation are returned as values of variables $v_{14}$ and $v_{15}$. These outputs are available on the output pins represented as put nodes in the demand graph. The demand graph gives also information about inter operation dependencies. The and operation, represented as node 12, can not start until the results of operations represented as node 9 and 11 are valid.

The input of the register-transfer level is the optimised demand graph. The hardware generator converts the demand graph into two descriptions, i.e. a description of the data path and a description of the state machine (= controller). During the synthesis of the data path and state machine a module library [Kaiser87] is used for retrieving information about pre-designed modules. A cost estimator is used for computing costs of implementations.
The lowest level is the layout level. The layout generator converts the symbolic description of the register-transfer level into a detailed layout description. Also placement and routing is done during this last phase. An exhaustive overview of the total silicon compiler can be found in [Jess87b].

1.2 THE HARDWARE GENERATOR.

Three basic interdependent tasks must be performed during the implementation of a demand graph. These tasks are:

1) Demand graph nodes are scheduled into machine cycles.
2) Demand graph nodes are mapped onto library modules.
3) Arcs of the demand graph are mapped onto interconnection nets between modules and/or register modules. A register is used each time the adjacent operations of an arc are assigned to different machine cycles.

The last two tasks can be grouped together as allocation. There are many ways to perform the hardware generator.
above tasks. Two methods are described in the next paragraphs. The first technique is based on dynamic programming. The main characteristic of this method is that scheduling and allocation are done concurrently. In figure 1.3a a pictorial overview of dynamic programming is given. The second method is based on divide and conquer techniques. The scheduling and allocation is done separately. First the scheduling is done, thereafter the allocation. In figure 1.3b a pictorial overview of the divide and conquer technique is given.

![Figure 1.2. Example of demand graph.](image)

I have used a method based on divide and conquer techniques. In paragraph 1.2.1 a short description of the dynamic programming technique is given. In paragraph 1.2.2 the divide and conquer technique is declared. In [Jess88] an overview of both methods is given.

![Figure 1.3. Overview of two techniques for hardware generation.](image)

The hardware generator.
1.2.1 Hardware generation using dynamic programming.

*Dynamic programming* can be defined as an optimisation strategy for a class of problems called multi-stage decision processes. The hardware synthesis problem can be described as a multi-step allocation problem. During the dynamic programming the nodes of the demand graph are implemented one by one. A node can only be implemented if it is *free*. A node is free if all related nodes have been implemented. For each implemented node a new state is created. A state is a set of implemented demand graph nodes. For each state several implementations can be made, for instance an existing piece of hardware can be used or a new one can be added to the implementation. Furthermore a choice has to be made between allocating the node in the current cycle or in the next one.

During the hardware generation the number of parallel 'live' states increases if no precautions are made. To reduce the number of parallel states the costs of comparable states are examined. Two states are comparable if the same set of demand graph nodes are implemented. The cheapest state of the comparable states is selected. All other comparable states are deleted. It is clear that the choice of the cost function influences the number of parallel states remarkably.

Some properties of hardware generation in case of using dynamic programming techniques are:

+ All tasks are done concurrently.
+ Many possible partial implementations are generated and examined.
- Huge search space leads to a memory and time consuming implementation.
- The implementation of a next state depends only on the current state. An overall scope of hardware requirements is not computed.

An exhaustive description of dynamic programming can be found in [Bellma62]. An implementation of a hardware generator based on dynamic programming techniques can be found in [Stok86] and [Wouden87].

1.2.2 Hardware generation using a divide and conquer technique.

To tackle the drawbacks of the dynamic programming technique a divide and conquer technique is used. The basic tasks performed by the hardware generator are divided into two mayor separated sub tasks. The first task consists of assignment of demand graph nodes to machine cycles. For the assignment of demand graph nodes a scheduler is used. At the current stage of the project the scheduler is under development. The second task consists of demand graph node allocation. The allocation of demand graph nodes can be divided further into three main phases. They are variable allocation, operation allocation and interconnection allocation.

*Variable allocation* deals with selection and implementation of registers and with assignment of variables to nets. For all variables used in more then one machine cycle a register must be implemented. To minimise the number of registers, variables can share a common register if two variables are not *live* at the end of in the same machine cycle.

*The hardware generator.*
Operation allocation deals with proper selection of modules on which demand graph nodes can be mapped. Several optimisation strategies can be formulated. One strategy could be: try to minimise the number of modules. This results in a small number of large and complex modules like alu's and adders. A better strategy could be minimise the total area used by modules, resulting in, maybe, more then the minimum number of modules.

Interconnection allocation deals with minimisation of interconnection nets. This is an important phase because in todays VLSI chips the total area used for interconnection between modules is 40% - 60% and this percentage is still growing. Multiplexing nets and use of register files decrease the number of nets.

If all three allocations are done the final implementation is found. It is clear that the final result heavily depends on the order of the allocations. To reduce the dependency of the allocation order, the allocations overlap somehow. During the operation allocation the number of interconnection nets to a module is hold as small as possible.

In the last five years several attempts have been made constructing hardware generators based on divide and conquer techniques, see for instance [Tseng86] and [Mei85]. The method used by Tseng and Mei and their minors shall be described shortly. The method of [Tseng86] is based on successively variable allocation, operation allocation and interconnection allocation. For all these allocations a heuristic clique search algorithm is used. Minors of this method are:

- There is no overlap of allocation phases.
- Expensive computation of variable allocation.
- No sequential operations (chaining) in one machine cycle.

The method of [Mei85] is based on successively operation allocation, interconnection allocation and variable allocation. The operations are allocated with use of Hu's algorithm that generates a sub-optimum solution for the case that the number of parallel operations is more than two. An advantage of this method, compared with the previous method is that the number of parallel operations can be defined by the user. For the interconnection allocation the A* algorithm is applied. The variable allocation is done similar as in the previous method. Minors of this approach are:

- Expensive computation of variable allocation.
- no overlap between allocation phases.
- Degrade of system performance during interconnection allocation as a result of the huge state space searching of the A* algorithm.
- No sequential operations (chaining) in one machine cycle.

For the hardware allocation used in the silicon compiler two methods are implemented. The first method consists of successive variable allocation, operation allocation and interconnection allocation. See for an overview figure 1.4a. The last two allocations overlap somehow. The second method is similar to the first method except the allocation of variables and allocation of operations are reversed. See for an overview figure 1.4b.

The operation allocation is based on grouping of operations together depending on the cost of combining them. The operation allocation can be translated into the problem of

The hardware generator.
finding a clique covering of a weighted graph. The variable allocation is based on a minimum colouring algorithm of an interval graph performed by a left edge algorithm. The interconnection allocation is also based on finding a clique covering of a weighted graph. The number of nets is reduced because of use of so called register files and multiplexing of nets.

Figure 1.4. Different order of operation and variable allocation.

The hardware generator.
2. PRE PROCESSING THE INPUT OF THE HARDWARE ALLOCATION.

2.1 THE SCHEDULER.

After the construction and optimisation of the demand graph a scheduler is used to assign demand graph nodes to timeslots. A timeslot is a coherent set of demand graph nodes scheduled into one machine cycle. An operation is a demand graph node and is represented as a symbol. For each operation at least one corresponding module must be available in the module library. An instruction consists of an operation and the operands of the operation. Thus if the demand graph node is meant then this is called an operation. If the operation inclusive the operands is meant then this is called instruction.

The operation assignment to timeslots depends on both area and time tradeoffs. An increase of time, i.e. the total number of timeslots is increased, leads to a decrease of required area. Multiple use of modules is easier so less modules are required. Analogous an increase of area, i.e. more parallel operations are scheduled into one timeslot, results in a decrease of time. During the scheduling the following decisions must be made:

- Number of parallel operations in a timeslot.
- Number of timeslots.
- Number of sequential operations (chaining) in a timeslot.

Decisions made during the scheduling are influenced by the constraints given by the user. The output of the scheduler is a description of the generated timeslots. The output is stored in a global vector called *time-slots*.

2.2 SYNTAX OF TIME-SLOTS.

Before the exact syntax in SBNF (Super-Backus Naur Form) is given, some rather trivial purposes of the *time-slots* are given. The number of all operations is equal to I, the total number of timeslots is T. The set of operations that are scheduled into the ith timeslot is Ti.

1) Sum of all operations Ti is equal to I, for 0 ≤ i ≤ T.
2) To avoid multiple assignment of operations into more then one Ti, the intersection between Ti and Tj, for 0 ≤ i,j ≤ T is empty.
3) At least one operation is scheduled in a timeslot Ti, i.e. Ti ≠ ∅ , for 0 ≤ i ≤ T,

The syntax of the *time-slots* can now be given in SBNF. Each entry of *time-slots* is a description of a single timeslot Ti. The description of a single timeslot Ti consists of four entries. The first entry, called <timeslot-name>, is a symbol name of Ti. The second entry, called <timeslot-operation-count>, is the number of operations in timeslot Ti. The third entry, called <timeslot-var-used-list>, is a list of variables that are live at the end

1. More information of modules and the module library can be found in paragraph 4.1.

**Syntax of time-slots.**
of \( T_i \). The last entry, called \(<\text{timeslot-data-count}>\), is used during the operation allocation. The initial value is nil. In SBNF the description is:

\[
\text{<*time-slots*>} ::= \text{"["} \text{<timeslot-name>}
                         \text{<timeslot-operation-count>}
                         \text{<timeslot-var-used-list>}
                         \text{<timeslot-data-count> \text{""]}".}
\]

\[
\text{<timeslot-name>} ::= \text{TimeSlot- <\text{number}> | <\text{symbol}>.}
\]

\[
\text{<timeslot-operation-count>} ::= \text{"\{"} \text{<variable> } \text{\} } \text{"\}".
\]

\[
\text{<variable>} ::= \text{<symbol>}. 
\]

\[
\text{<number>} ::= \text{<integer>}. 
\]

\[
\text{<timeslot-data-count>} ::= \text{"nil" | <\text{number}>.}
\]

In figure 2.1 a pictorial example of \(*\text{time-slots}*\) is given.

![Diagram](image-url)

**Figure 2.1.** Pictorial example of \(*\text{time-slots}*\).

For the example of 2.1 the description of the \(i\)th timeslot is:

\[
\text{<timeslot-name>} : T_i. 
\text{<timeslot-operation-count>} : 3. 
\text{<timeslot-var-used-list>} : \{V3,V4,V5\}. 
\text{<timeslot-data-count>} : \text{nil.}
\]

### 2.2.1 Description of a single timeslot.

As said before, the first entry of a timeslot description is the \(<\text{timeslot-name}>\). The value of this symbol is a vector called \textit{instruction-vector}. An \textit{instruction-vector} consists of

*Syntax of time-slots.*
instructions. An instruction consists of an operation and the operands of the operation. Operands can be: first-operand, second-operand or result-operand. For the divide operation in timeslot $T_i$ in figure 2.1 the instruction format is:

\[
\begin{align*}
&\text{operation} : ~/ \text{.}
&\text{first-operand} : V0.
&\text{second-operand} : V1.
&\text{result-operand} : V3.
\end{align*}
\]

If an operation is monadic then the second-operand shall be not specified and therefore nil. The syntax of a get and a put instruction is different. For a get instruction the first and second operand are not specified. For a put instruction the result and second operand are not specified.

It is possible to use a memory. The memory can be implemented on chip or off chip. An external memory is an off chip memory. The operations of an on chip memory are read and write. The operations of an off chip memory are mread and mwrite. For both types the same instruction syntax is used. The first entry, called <memory-operation>, is the operation that must be performed. The second entry, called <memory-name>, is the name of the used memory. The third entry, called <index-name>, is a value used for proper selection of the accessed memory address. The fourth entry, called <data-name>, gets the value of the accessed memory address in case of a read or mread operation. If a write or mwrite operation is performed the accessed memory address gets the value of <data-name>.

The instruction-vector syntax in SBNF notation of a timeslot is:

\[
\begin{align*}
&\text{instruction-vector} ::= "\{ \text{instruction} \}^* "\].
&\text{instruction} ::= \text{operation-instruction} | \text{get-instruction} | \text{put-instruction} | \text{memory-instruction}.
&\text{operation-instruction} ::= \text{operation} \text{ result-operand} \text{ second-operand} \text{ second-operand} \text{].}
&\text{get-instruction} ::= \text{GET <variable> nil nil "\].}
&\text{put-instruction} ::= \text{PUT nil <variable> nil "\].}
&\text{memory-instruction} ::= \text{memory-operation} \text{ memory-name} \text{ index-name} \text{ data-name} "\].
&\text{operation} ::= <symbol>.
&\text{result-operand} ::= <variable>.
&\text{first-operand} ::= <variable> | <constant>.
&\text{second-operand} ::= <variable> | <constant> | "nil".
&\text{memory-operation} ::= READ | MREAD | WRITE | MWRITE.
&\text{index-name} ::= <variable> | <constant>.
&\text{data-name} ::= <variable> | <constant>.
&\text{variable} ::= <symbol>.
&\text{constant} ::= <integer>.
\end{align*}
\]

In appendix I the syntax of *time-slots* and the syntax of the contents of one timeslot are given. For an overview of operations currently performed see table 4.1 in paragraph 4.1.
2.3 PRE-PROCESSING TIME-SLOTS.

Before the allocation can be done a pre-processor is used. The task of the pre-processor is to initialise all global variables and to extract and compute useful information from *time-slots*. Tasks performed by the pre-processor are:

- Selection of pad drivers and multiplex modules.
- Create list of all variables that must be stored into registers.
- Replace constants by constant names and create constant database.
- Creation of port database and memory database.

In this paragraph these different tasks are explained further.

Selection of pad drivers and multiplex modules.

Pad drivers are used for allocation of *get* and *put* operations. Three different kinds of pads can be distinguished: an input pad, an output pad and an input/output pad. Selection of a proper module for these three kinds of pads is done during the pre-processing. Multiplex modules are used for selecting the proper input of a multiple used module. Selection of proper multiplex modules with various number of inputs is done during the pre-processing.

Create list of variables that must be stored into registers.

During the variable allocation it is important to know if a variable must be stored into a register. A variable must be stored into a register if a variable is *live* in more then one timeslot. A variable $x$ is *live* from its first definition till the last use of it. A variable $x$ that is *live* in more then one timeslot is called a *store variable*. A variable $x$ is an *internal variable* if $x$ is only used in one timeslot.

All store variables together form a list called •variables-to-store-list*. The property *get-internal-edge-value* of all internal variables is set true. The default value is nil.

Replace constants by constant names and create constant database.

In an instruction a constant (integer value) can be used as first or second operand of an operation. For a constant a module must be selected which can deliver the desired constant value at the output of the module. For proper function in the allocation phases the constant is replaced by a unique symbol representing the constant. For retrieving the original value of a constant and the generated symbol name, a constant is stored into a so called constant database.

Creation of port database and memory database.

The *get* operation and *put* operation use ports for receiving or transmitting data from or to the outside of the chip. A port can be an input port, an output port or an input/output

---

2. This is necessary because in Lisp no properties can be given to integers.

*Pre-processing time-slots.*
port depending on the operations performed by a port. Each port is implemented as a module. To store the behaviour of a port a port database is maintained. For each port the port name, the variables used as input of a port and the variables used as output of a port are stored. If the input list or the output list is not used then the corresponding entry in the port database becomes nil.

It is possible to use a memory in an algorithm. The memory can be implemented on_chip or off_chip. For both types some information must be maintained. This information is stored in a memory database. This information is:

- The name of the memory.
- The variables used for indexing the accessed memory address.
- The 'data names' used for retrieving or storing values in the memory.
- The type of the memory, i.e. on_chip or off_chip.

All tasks performed by the pre-processor are declared. The syntax of databases used for memory, constants and ports is given in appendix 2. For completeness the algorithm of the pre-processor is given on the next page.

Throughout this report several algorithms are described. The syntax of all these descriptions is equal.

- bold style is used for function names.
- italic style is used for variables.
- roman style is used for text.

Pre-processing time-slots.
The pre-processor algorithm

local $i,j,\text{operand}$

init-global-variables
init-multiplex-cost
select-pad-drivers-and-multiplexers
for $i = 1$ to $T$
  do
    for $j = 1$ to $D_i$
      do
        for operand = ($\text{result-operand}_j$, $\text{first-operand}_j$, $\text{second-operand}_j$)
          if internal-variable(operand)
            get-internal-edge-value(operand) := true
          fi
          if constant-detected(operand)
            update-constant-database(operand)
          fi
        od
      od
    update-variables-to-store-list($T_i$)
    if port-used-detected($\text{operation}_j$)
      update-port-database($\text{operation}_j$)
    fi
    if memory-used-detected($\text{operation}_j$)
      update-memory-database($\text{operation}_j$)
    fi
  od
rof

with $T$: total number of timeslots.
$T_i$: one timeslot description.
$D_i$: the number of operations in one timeslot $T_i$.

Pre-processing time-slots.
3. VARIABLE ALLOCATION.

As stated in the previous chapter two types of variables can be distinguished. These types are internal variables and store variables. The internal variables are mapped onto interconnection nets between modules. The store variables are mapped onto register modules. Multiple use of one register for storage of several variables reduces the number of required registers and therefore the used area. Store variables can be stored into the same register if the variables are not live at the same time. The problem of store variable allocation is: find a minimum number of registers holding all store variables, such that no variables are live at the same time in one register. In the rest of this chapter with variable allocation is meant the allocation of store variables.

In paragraph 3.1 it is shown that the variable allocation problem can be translated into a minimum colouring of an interval graph. The colouring of an interval graph is rather simple because an interval graph is a perfect graph. The colouring can be done with the so-called left edge algorithm used in channel routers. In paragraph 3.2 a proof is given based on the minimum colouring of a chordal graph.

3.1 USE OF LEFT EDGE ALGORITHM FOR VARIABLE ALLOCATION.

The left edge algorithm computes an optimal solution for the minimum colouring of the graph nodes if the graph is an interval graph. An undirected graph $G(V,E)$ is an interval graph if there is a linearly ordered set of intervals; the vertices $V$ of the graph $G$ are the intervals and there is an edge $E$ between two vertices if and only if the two corresponding intervals intersect. Two intervals intersect when they overlap.

Define the variable interval graph, $VIG(V,E)$ as a graph with vertices $V$ of the graph $VIG$ are the variables and there is an edge $E$ between two vertices if and only if the two corresponding variables are live at the end of a timeslot $T_i$. The constructed graph $VIG$ is an interval graph.

EXPLANATION

A variable is said live from its definition till the last use. The time that a variable is live can be modelled as an interval in time. The interval starts at the definition of a variable and ends on the last use of it. For each variable there is one continuous interval. If two intervals overlap in at least one time point then they intersect. An example of intersecting intervals is given in figure 3.1. In figure 3.1a each variable is represented as a horizontal line. The beginning of a line is the moment of the variable definition; the end of a line is the moment of its last use. Remark: only the store variables are depicted.

It is clear that the intersecting variables form a linearly ordered set of intervals and therefore can be represented by an interval graph.

3.1.1 Construction of variable interval graph.

Due to the scheduler operations are scheduled into timeslots. The corresponding variables of operations are therefore also scheduled into timeslots. A possible scheduling of

Use of left edge algorithm for variable allocation.
variables given in figure 3.1a is given in figure 3.1b. The variables that are live at the end of a timeslot must be stored into a register. For each intersecting pair of variables who are both live at the end of a timeslot, an edge is added to the variable interval graph. The resulting variable interval graph is depicted in figure 3.2.

Figure 3.2. Resulting variable interval graph of example.

The nodes of the interval graph VIG can now be coloured with the left edge algorithm. The left edge algorithm works only correct if the intervals are ordered on the left edge value. The intervals of the interval graph VIG are automatically ordered so no order must be computed. It is easy to understand that the maximum number of different colours is equal to the maximum number of intersecting intervals in a timeslot.

Working left edge algorithm:

Let \( \{i_1,i_2,\ldots,i_N\} \) the ordered set of intervals. The total number of intervals is \( N \). Let \( \{c_1,c_2,\ldots,c_C\} \) the set of used colours. The total number of colours is \( C \).

1) Assign \( i_1 \) to \( c_1 \).

---

3. It is easy to detect if two variables intersect because in "timeslots" the field <timeslot-var-used-list> is maintained.

Use of left edge algorithm for variable allocation.
2) Assuming that \( i_1, i_2, \ldots, i_{k-1} \) are already assigned to colours, assign \( i_k \) to an arbitrary available colour.

The availability of a colour is guaranteed by the following argument. If \( i_k \) can not be assigned to any colour then the left edge of \( i_k \) intersects with previously assigned intervals with colours \( c_1, \ldots, c_M \). This brings the total density to \( C+1 \). But the maximum number of intersecting intervals was \( C \), so \( i_k \) can not intersect with previously assigned intervals with colours \( c_1, \ldots, c_M \).

Ordering the intervals given in figure 3.2 on the left edge results in \( \{v_1, v_4, v_2, v_5, v_3, v_6\} \). The number of intervals is 6. The maximum number of intersecting intervals is 3. After applying the left edge algorithm, we get:

\[
\{v_1, v_5\} \\
\{v_2, v_3\} \\
\{v_4, v_6\}
\]

Each set of variables are mapped onto one register.

### 3.2 MINIMUM COLOURING OF AN INTERVAL GRAPH WITH USE OF A R-ORIENTATION.

A chordal graph is an undirected graph in which every cycle of length exceeding 3 has a chord, i.e., an edge joining two nonconsecutive vertices in the cycle. Gavril and others [GavrIl72], [Gilmor64] have proved that an interval graph is a special class of the chordal graphs. Gavril uses the R-orientation of the graph nodes to find a minimum colouring.

An orientation of the graph its nodes is called a R-orientation if the following two conditions are satisfied:

(i) The resulting directed graph has no directed cycles. 
(ii) If \( b \rightarrow a \) and \( c \rightarrow a \), then \( b \rightarrow c \); that is, either \( b \rightarrow c \) or \( c \rightarrow b \).

For computation of a R-orientation the edges of an interval graph are divided into so called sections. The first section starts at the beginning of the start of the leftmost interval. A section ends if an interval in the section ends. The total number of sections is \( P \). A section has the following properties:

1) At least one interval \( i_k \), \( 0 \leq k \leq N \), is member of a section \( S_i \), \( 0 \leq i \leq P \).
2) the total number of sections is \( P \leq N \).

The sections for the example given in figure 3.1 are:

\[
S_1 = \{v_1, v_4\} \\
S_2 = \{v_2, v_4\} \\
S_3 = \{v_2, v_5\} \\
S_4 = \{v_3, v_5, v_6\} \\
S_5 = \{v_5, v_6\}
\]

The R-orientation of the nodes of the corresponding interval graph can be computed easily. Start with the first section and add all intervals to the R-orientation list. Then the second section is taken. All intervals of this section are added to R-orientation list. This
process is repeated until all sections have been examined. Thereafter the R-orientation list
is reversed. All duplicates of intervals are removed. For the example the R-orientation
list becomes:

\[ R\text{-orientation list} := \{v_6, v_5, v_3, v_2, v_4, v_6\} \]

The direction of the interval graph edges is based on the R-orientation list. Start with the
first vertex in R-orientation list. The edge direction of edges connected to the first vertex
gets a direction to the adjacent vertices. The first vertex is deleted from R-orientation
list. Above process is repeated until no edges left in R-orientation list.

Has the constructed graph a R-orientation?

To answer this question, the conditions of equation (3.1) must be satisfied.

Condition 1 Suppose there is a cycle \(A \rightarrow B \rightarrow \ldots \rightarrow A\), and \(A, B, C\) is the order in R-
orientation list. This is not possible because the edge to \(A\) cannot appear because \(A\) is
examined earlier than all other vertices. So the direction of an edge connected to \(A\) is
always to the adjacent nodes and therefore no cycle can occur.

Condition 2 Suppose this condition is not valid. Then the following structure occurs.
There is an edge from \(A\) to \(C\), and there is another edge from \(B\) to \(C\). There is no edge
between \(A\) and \(B\). This situation is depicted in figure 3.3.

Let the position of a vertex in the R-orientation list be given by \(P(\text{vertex})\). If vertex \(A\)
appears earlier in the R-orientation list then vertex \(B\) then \(P(A) < P(B)\). Now examine the
situation depicted in figure 3.3. Because there is an edge from \(A\) to \(C\), \(P(A) < P(C)\) and
there must be a section \(S_i\) with \(A\) and \(C\) in it. The same argument holds for \(B\), \(P(B) < P(C)\)
and a section \(S_j\). Because there is no edge between \(A\) and \(B\) \(i \neq j\). Two R-orientations can
now occur \(P(A) < P(B)\) or \(P(B) < P(A)\).

Suppose \(P(A) < P(B)\).

Then \(P(A) < P(B) < P(C)\). \hspace{1cm} (3.2)

For \(i < j\) and find a R-orientation results in \(P(C) < P(B) < P(A)\).
For \(i > j\) and find a R-orientation results in \(P(C) < P(A) < P(B)\).

Minimum colouring of an interval graph with use of a R-orientation.
But for an invalid condition 2 equation (3.2) must be found.

Suppose $P(B) < P(A)$.

Then $P(B) < P(A) < P(C)$. (3.3)

For $i < j$ and find a R-orientation results in $P(C) < P(B) < P(A)$.
For $i > j$ and find a R-orientation results in $P(C) < P(A) < P(B)$.

But for an invalid condition 2 equation (3.3) must be found. Therefore condition 2 is always valid so the constructed directed interval graph has a R-orientation.

3.2.1 Minimum colouring algorithm for the variable interval graph.

The resulted directed interval graph of the previous paragraph must be coloured. This can be done easily with use of the R-orientation list. The colouring goes as follows:

1) The first vertex of R-orientation list is taken. The colour of this vertex becomes $c_1$.

2) Suppose the first $k-1$ vertices of R-orientation list are coloured. And the resulting colouring is a minimum colouring. Suppose the number of colours is $M$. Then the next vertex $k$ is taken from R-orientation list. $K$ is added to set of the first colour found without destroying its independence (meaning no edges between vertices of a set). If no such set can be found a new colour is generated. And $k$ is added to this new set.

EXPLANATION

Consider the first vertex $v$ who gets a new colour $c_1$. It was added to the new set because $v$ was connected to vertices which have all a colour $c_j$, with $j < i$. Since all these edges are directed to $v$ (else $P(v)$ was not correct in R-orientation list) there is a clique with size $i$. Therefore, any colouring must have at least $i$ colours and hence the found colouring is a minimum colouring.

3.3 DISPATCH LOOPS AND SUB ROUTINE CALLS DURING VARIABLE ALLOCATION.

In the high level input description loops and/or subroutines are allowed. Both use of loops and subroutines influences the variable allocation. Until now each variable can be represented as one continues time interval. But due to the use of loop variables\(^4\) this assumption is no longer valid. A loop variable is a variable that is used in a loop. A loop variable can be defined more then once and is thus represented as a not continues interval. An example of this situation is depicted in figure 3.4. The variable $v_2$ is defined two times, once in timeslot $T_1$ and once in $T_3$. Therefore it is not possible to create an interval graph for the variables of figure 3.4 because the corresponding variable

\(^4\) The same holds for variables used in sub routine calls.
intervals cannot be sorted linearly. The simple variable allocation of the previous paragraphs cannot be used. To tackle this problem the representation of the variables is changed.

Suppose the set of store variables is \( \{v_1, \ldots, v_M\} \) with \( M \) the total number of store variables. For each variable \( v_n \), \( 0 \leq n \leq M \), the number of intervals is examined. Suppose that for a variable \( v_n \), that consists of more than one interval the set of intervals is \( \{i_{n1}, \ldots, i_{nK}\} \), with \( K \) the number of intervals. Then for each interval \( i_{nk} \) an unique sub variable name is generated. The variable name \( v_n \) is deleted from the set of store variables and the generated sub variable names are added to it. Suppose that the new variable names are \( \{v_{n1}, \ldots, v_{nK}\} \). Then the resulting store variable set is \( \{v_1, \ldots, v_{n1}, \ldots, v_{nK}, \ldots, v_M\} \). For variables of figure 3.4 the resulting variable set is given in figure 3.5.

The variable \( v_2 \) is represented as two variables, i.e. \( v_{21} \) and \( v_{22} \). It is easy to understand that the new set of store variables can be represented as an interval graph and therefore the previous used method of variable allocation can be used.

Although the variable allocation can be done fast a problem occurs during the allocation. A variable \( v_n \) is replaced by a set of sub variables. It is not guaranteed that the all sub variables of a variable \( v_n \) are grouped together in one register. Therefore an operation is added to the set of operations. This operation is called trans operation. The corresponding trans instruction consists of two operands. A trans operation transfers the value of a register used for storage of a variable \( v \) to the register used for storage of a variable \( v' \). Due to the use of trans operations it is possible to use the same register if a variable \( v_n \) is

 Dispatch loops and sub routine calls during variable allocation.
represented by sub variables.

For the variable \( v_2 \) of the example this goes as follows. Suppose the sub variable \( v_{21} \) is stored into register \( R_2 \) and the sub variable \( v_{21} \) is stored into register \( R_1 \). In timeslot \( T_1 \) register \( R_2 \) is used for variable \( v_2 \). Before timeslot \( T_3 \) the value of \( R_1 \) must be transported with use of a trans operation to register \( R_2 \).

The trans operation is implemented as an extra interconnection from the output of a register to the input of another register (or register input multiplexer).

Remarks:

- The trans operation works only correct for transport of store variables.
- If two sub variables \( v_{ij} \) and \( v_{ij} \) are stored into the same register the additional extra interconnection net is unnecessary.

Dispatch loops and sub routine calls during variable allocation.
4. OPERATION ALLOCATION.

In this chapter the operation allocation is detailed further. In paragraph 4.1 some aspects of the module library are explained. In paragraph 4.2 the data operation allocation is given. In paragraph 4.3 the method used to solve the operation allocation is explained. The method is based on finding a clique cover of a weighted graph.

4.1 THE MODULE LIBRARY.

As described earlier the hardware generator heavily relies on a module library. A module library consists of a finite set of modules. A module is a functional unit that can perform one or more operations. The module library should store two kinds of data, data describing the modules and data describing operations. For each operation in the input description a corresponding library module must be provided in the library. New operations used in the input description can be added easily to the library.

For both operations and modules aspects are defined. Examples of aspects are commutativity, representation, monadic or dyadic for operations and area, delay, name, inputs, outputs and control for modules.

4.1.1 Target architecture.

Currently the target architecture of the silicon compiler consists of a data path description and a state machine description. The data path consists of a set of modules performing operations, mutual connected with interconnection nets. Both single and multiple operation modules are allowed. At this stage of the silicon compiler development a multiplex design style is implemented. This means that the value of a net only depends on the output of one module. If the inputs of a module steam from different sources a multiplexer is needed at the input of a module for proper selection of an input source. Both data path and state machine are synchronised with a clock. No self timed circuits are allowed.

The modules in the module library can be divided into three major groups. These groups are:

- storage modules.
- operation modules.
- data-routing modules.

A storage module is a module that can store some value. A storage module has a control input for activating the storage of a new value. Examples of storage modules are registers and register files. The last one consists of at least two registers grouped together into one module. A register file needs an additional control input for proper selection of a register in the register file. Some storage modules must always be provided in a module library.

*The module library.*
An *operation module* is a module that performs an operation specified in the input description. The number of operation modules depends on the operations used in the input description. An example of a possible set of operations is given in table 4.1.

A *data-routing module* is a module that directs data flow between operation modules. The output of a data-routing module depends on the control input. Examples of data routing modules are multiplexers, de-multiplexers and busses.

### TABLE 4.1. Example of implemented operations in the library.

<table>
<thead>
<tr>
<th>Types of operations</th>
<th>data-operations</th>
<th>transfer-operations</th>
</tr>
</thead>
<tbody>
<tr>
<td>operation</td>
<td>commutative</td>
<td>dyadic</td>
</tr>
<tr>
<td>+</td>
<td>t</td>
<td>t</td>
</tr>
<tr>
<td>-</td>
<td>f</td>
<td>t</td>
</tr>
<tr>
<td>*</td>
<td>t</td>
<td>t</td>
</tr>
<tr>
<td>/</td>
<td>f</td>
<td>t</td>
</tr>
<tr>
<td>square</td>
<td>-</td>
<td>f</td>
</tr>
<tr>
<td>sqrt</td>
<td>-</td>
<td>f</td>
</tr>
<tr>
<td>=</td>
<td>t</td>
<td>t</td>
</tr>
<tr>
<td>&gt;</td>
<td>f</td>
<td>t</td>
</tr>
<tr>
<td>&lt;</td>
<td>f</td>
<td>t</td>
</tr>
<tr>
<td>≥</td>
<td>f</td>
<td>t</td>
</tr>
<tr>
<td>≤</td>
<td>f</td>
<td>t</td>
</tr>
<tr>
<td>and</td>
<td>t</td>
<td>t</td>
</tr>
<tr>
<td>or</td>
<td>t</td>
<td>t</td>
</tr>
<tr>
<td>nor</td>
<td>t</td>
<td>t</td>
</tr>
<tr>
<td>not</td>
<td>-</td>
<td>f</td>
</tr>
</tbody>
</table>

### 4.1.2 The operation modules in the module library.

In the previous paragraph the modules are divided into three groups. The operation modules can be divided further into two subgroups. The first subgroup consists of *transfer operation* modules. These modules take care of the data transfer. An example of a transfer operation module is a pad-driver. Some transfer modules are always in the library. The other subgroup, called *data operation* modules, consists of modules performing data operations. For each specified operation a data operation module is provided in the library.

To retrieve information from the module library a library interface is defined. The library interface is a part of the module library. Both information about operations and modules can be retrieved. During the operation allocation the following aspects are used:

---

5. See chapter 5 for more information about register files.
The aspect area is used to select a module which uses the minimum amount of area to perform a specific operation.

- The aspect delay is used to optimise time requirements during the operation allocation.

- The commutativity of operations is used to optimise the number of interconnection nets. The notion of commutativity can be extended to modules. A module is commutative if the performed operations are commutative.

- The aspect operations of a module is used to find out if different operations can be mapped onto one multi-operation module. Examples of a multi-operation modules are an alu or an adder.

An exhaustive overview of the module library used in the silicon compiler is given in [Kaiser87].

4.2 DATA OPERATION ALLOCATION.

The operation allocation is divided into the allocation of transfer operations and the allocation of data operations. The transfer operation allocation can be done easily and is specified by the user. For each variable used in a transfer operation the ports used for operands are specified in the input description. For each transfer operation the proper transfer operation module is selected from the module library. No optimisation can be done because transfer operations can not be combined. In contrast to the transfer operation allocation the data operation allocation can be optimised. In this paragraph the data operation allocation is described. First the cost functions used during the data operation allocation are given.

4.2.1 Cost functions used for data operation allocation.

Before the cost functions are given a definition of a mod is given. A mod is a complete description of a set of data operations. The initial mod description describes one data operation. A mod description consists of 7 separated fields. These fields are:

- The operation set OS of a mod X is the set of operations that are mapped onto the same module.

- The timeslot set TS of a mod X is the set of timeslots in which a mod is active. A mod is active if it is used in a specific timeslot.

- The result set RS of a mod X is the set of variables or register names that are used as result operand of the operation set OS(X).

- The in sets ISO and ISI of a mod X are the sets of variables or register names used as input operands to the operation set OS(X). ISO is the set of the leftmost input operands, ISI is the set of the rightmost input operands. If there are no dyadic operations in OS(X) then ISI becomes nil.

- The library module name of a mod X is the name of the library module used to allocate OS(X).

- The module cost of a mod X is the cost of the used library module. The cost is computed with cost function cf1.

Data operation allocation.
The multiplex cost of a mod $X$ is the cost of the multiplexers used for proper selection of the inputs of a module.

During the operation allocation two cost functions are used. The first cost function $cf_1$ is used to compute the cost of a library module. The second cost function $cf_2$ is used to compute the cost of a mod pair. In the next sub paragraph both functions $cf_1$ and $cf_2$ are explained further.

4.2.1.1 Description of $cf_1$ used for cost estimation of a library module.

The function $cf_1$ is used to compute the cost of a library module. Currently two estimations are implemented. The first one uses only the area of a module library. The second one uses both area and delay of a module. The last one is:

$$cf_1(module) = \text{delay}(module) \times \text{area}(module).$$  \hspace{1cm} (4.1)

By proper setting of the adjust vector one of the implemented versions of the cost function $cf_1$ can be selected.

4.2.1.2 Description of $cf_2$ used for cost estimation of mod pairs.

The cost function $cf_2$ estimates the cost of combining two mod descriptions. The following considerations are made:

Modules in the module library.

When two mods, $X$ and $Y$, with corresponding operation sets $\text{OS}(X)$ and $\text{OS}(Y)$ are combined into a new mod $Z$ then there must be a library module that can perform the union of $\text{OS}(X)$ and $\text{OS}(Y)$. If there is no such module then mod $X$ and mod $Y$ can not be combined and therefore $cf_2$ is infinite.

Operation set $\text{OS}$.

The cost of two mods, $X$ and $Y$, that have a common operation in $\text{OS}(X)$ and $\text{OS}(Y)$ should be less then the cost of $X$ and $Z$ if the intersection of $\text{OS}(X)$ and $\text{OS}(Z)$ is empty.

Timeslot set $\text{TS}$.

When two mods, $X$ and $Y$, are active in the same timeslot $T_i$ then $X$ and $Y$ can not be combined. This is done because the total delay in a timeslot changes when two mods in one timeslot are combined. Also the implementation of the controller is difficult because sub states within a state must be generated. An example is given in figure 4.1.

There are two mods, $\text{mod}_0$ and $\text{mod}_1$ in figure 4.1a. Suppose the delay of both modules is $D$. Then in 4.1a the delay in timeslot $T_i$ is $D$. If both modules are combined into one module, named $\text{mod}_01$ (see 4.1b) the total delay time is $2D$ and thus increased. The multiplexer $\text{mux}$ needs two times a control signal in one timeslot because first the leftmost input of $\text{mod}_0$ must be selected and thereafter the leftmost input of $\text{mod}_1$.

Data operation allocation.
Figure 4.1. Example of increase of delay.

Result set RS.

If two mods, X and Y, 'write' their results to the same register module or to the same operation module then it is preferable to combine X and Y into one mod instead of a combination of mods that write to different destinations. In the first case the number of interconnection nets is reduced by one.

In sets ISO and ISI. If two mods, X and Y, have a common source then the costs of combining X and Y should be less then the combination of X and Z if X and Z have not a common source.

Multiplex cost.

In [Tseng86] only the costs of the in sets and result set of two mods are used when two mods are combined. A better result can be achieved if not only these costs are taken into account but also the costs of the required multiplexers at the input of a module. Therefore the multiplex cost is a part of the total cost of a mod. The total cost of a mod is the sum of the module cost and the multiplex cost.

\[
\text{total\_cost\_mod}(X) = \text{module\_cost}(X) + \text{mux\_cost\_mod}(X) \quad (4.2)
\]

Both \text{module\_cost} and \text{mux\_cost\_module} are computed with \( cf_1 \). The mux cost of a mod is the sum of three costs, cost for ISO, ISI and RS respectively. For each set the number of different nets is computed. If this number is N then a multiplexer with at least N inputs is needed. Then the library is searched through for a multiplex module with at least N inputs. If there exists such a module then the cost of that module is computed with \( cf_1 \). If there is no such module then the multiplex cost becomes \textit{infinite}.

Although only multiplexers are required at the inputs of a module, the costs of the result set RS is part of the multiplex cost. This is done because if a module 'writes' to different modules then the number of nets is increased. Therefore the cost is also increased\(^6\). Thus mux costs of a mod becomes:

\[\quad\quad\]

\(^6\) The setting of the cost function can be changed such that the cost of the result set is not part of the multiplex cost. For \( K = 1 \) in (4.3) no costs for the result set are added to the mux cost of a module.

\[\quad\quad\]

\textit{Data operation allocation}. 

The different properties of \( cf_2 \) are discussed. Now the procedure for computing \( cf_2 \) can be given.

Procedure compute\(_{cf_2}(X,Y)\)

```plaintext
local Z, cost
if combine_mod_p(X,Y)
    then do
        Z := find_minimum_combined_mod(X,Y);
        if (< total_cost_mod(Z)
            (+ total_cost_mod(X)
                total_cost_mod(Y))
            then cost := compute_tseng_cost(X,Y);
            else cost := max_cost;
        fi;
    od;
else cost := infinite;
fi;
```

\( X, Y \) and \( Z \) are descriptions of mods. The function `combine_mod_p(X,Y)` tests if the mods \( X \) and \( Y \) are active in the same timeslot. If this occurs then `combine_mod_p` returns nil and the cost becomes `infinite`. If the mods of \( X \) and \( Y \) are not active in the same timeslot a combined mod, called \( Z \), is generated. The function `find_minimum_combined_mod` combines the sets \( OS, TS, RS, IS0 \) and \( IS1 \) of \( X \) and \( Y \). Also the module name, module cost and multiplex cost are computed. The function scans through the library for a list of modules that can preform the union of the operation sets of \( X \) and \( Y \). If no such module exists then the module cost becomes `infinite`, else the module with the minimum cost is selected.

The function `total_cost_mod` computes the total estimated cost of a module as described in equation 4.2. The cost becomes equal to `max_cost` if it is not beneficial to combine \( X \) and \( Y \) in spite of that \( X \) and \( Y \) are not active at the same time; the total cost of \( Z \) is greater then the sum of the total costs of \( X \) and \( Y \).

If it is preferable to combine \( X \) and \( Y \) into a new mod \( Z \) then the function `compute_tseng_cost` is called. This function computes the cost of combined mods similar as in [Tseng86]. For completeness the eight cost categories described by [Tseng86] are given in Figure 4.2. Remark: In [Tseng86] the categories starts with \( G7 \) for best combination till \( G0 \) for the worst combination of mods.

### 4.3 CLIQUE COVER OF DATA OPERATION GRAPH.

To optimise the data operation allocation a graph is constructed. This graph is called *data operation graph* \( DOG(V,E) \). The node set \( V \) of \( DOG \) consists of mods. The edge set \( E \) consists of mod pairs that can be combined. The weight of an edge in \( E \) is a measure for the profit of combining two mods and is computed with the cost function \( cf_2 \).

_Clique cover of data operation graph._
G0: The operations and three pairs of variables are all the same.

G1: The operations are different but the three pairs of variables are the same.

G2: The operations and two pairs of variables are the same, one pair of variables is different.

G3: Two pairs of variables are the same, the operations and one pair of variables is different.

G4: The operations and one pair of variables are the same, two pairs of variables are different.

G5: One pair of variables is the same, two pairs of variables and the operations are different.

G6: The operations are the same, the three pairs of variables are different.

G7: The operations and the three pairs of variables are different.

Figure 4.2. Cost according to Tseng.

4.3.1 Initial structure of data operation graph.

For each operation a mod description is generated. The seven fields of a mod description are loaded with the proper values. The initial value of the multiplex cost of a mod is 0. The library module name is the name of the module which performs the operation and the total cost is minimal. An example of the graph DOG is given in figure 4.3.

The data operation allocation is implemented as a clique cover of the graph DOG. The cliques of the graph DOG must have the following purposes:

1) The cost of a clique $C_i$ is minimal.

2) The costs between two cliques $C_i$ and $C_j$ is maximal.

A clique is a complete connected (sub) set of graph nodes. So there is an edge between each node pair of the clique. The edges of a clique are called internal clique edges. The edges between cliques are called inter clique edges. The cost of a clique $C_i$ is the sum all its internal clique edges. The costs between two cliques is the sum of all inter clique edges connected to the two cliques.

To solve above problem several strategies can be made. It shall be clear that the exact computation of the data operation allocation is NP-complete, i.e. finding a clique cover of a graph is normally NP-complete [Garey79]. To achieve a near optimum solution in a reasonable amount of time a heuristic is used. From the graph DOG a subgraph is extracted. This graph is called sub operation graph SOG($V', E'$). The node set $V'$ consists of mods. The edge set $E'$ consists of edges which have the lowest weight in DOG. An example of SOG is given in figure 4.3. Graph SOG construction can be done in linear time equal to the number of edges in DOG. The node numbers in DOG and SOG are the corresponding numbers of mods. Note that the edges in SOG do not have a weight. To find a clique cover of the DOG graph the following heuristic is used.

---

7. If the cost of an edge $(X,Y)$ is infinite then no edge between $X$ and $Y$ is depicted in figure 4.3.

Clique cover of data operation graph.
Figure 4.3. Example of data operation graph \( DOG \) and sub operation graph \( SOG \).

**Heuristic 1:**

The initial structure of \( DOG \) is computed. Thereafter the sub operation graph \( SOG \) is extracted from \( DOG \). For \( SOG \) a minimum clique cover is computed. Suppose that the resulting cliques are \( C_1, C_2, \ldots, C_N \). For each clique \( C_i \), \( 1 \leq i \leq N \) the graph \( DOG \) is updated. Suppose \( C_1 \) consists of \( \{n_1, n_2, n_3\} \). Then the nodes and the connected edges of \( n_2 \) and \( n_3 \) are deleted in \( DOG \). The corresponding mod descriptions of \( n_2 \) and \( n_3 \) are combined with the mod description of \( n_1 \). The edge weight of edges connected to \( n_1 \) are recomputed with \( cf_2 \). If all cliques \( C_i, 1 \leq i \leq N \), are examined the next \( SOG \) is extracted from the reduced \( DOG \). This process is repeated until there are no edges left in \( DOG \).

The above approach reduces the computation time because there are normally only a small number of nodes in \( SOG \). Therefore a minimum clique cover can be computed in a reasonable amount of time. To reduce the computation time further a second heuristic is used for the computation of a minimum clique cover of \( SOG \). This heuristic is described in the next paragraph and is similar with the method used in [Tseng86].

The algorithm of data operation allocation is:

\[
\text{Data operation allocation.}
\]

\[
\text{local } DOG, SOG
\]

\[
DOG := \text{create-data-operation-graph}
\]

\[
SOG := \text{select-sub-operation-graph}(DOG)
\]

\[
\text{while (not (null SOG))}
\]

\[
\text{find-minimum-clique-cover(SOG)}
\]

\[
\text{update-data-operation-graph}(DOG)
\]

\[
SOG := \text{select-sub-operation-graph}(DOG)
\]

\[
\text{endwhile}
\]

4.3.2 Minimum clique cover of sub operation graph \( SOG \).

The problem of computing a minimum clique cover is: divide the nodes of a graph in a minimum number of disjoint sets \( C_i \), such that each node is in one and only one subset \( C_i \).

\[
\text{Clique cover of data operation graph.}
\]
and the nodes of a subset $C_i$ form a complete connected graph.

As stated before the problem of finding a minimum clique cover of a graph is NP-complete. To avoid long computation time a heuristic is used. The heuristic is based on the neighbourhood property among nodes to partition a graph into a set of disjoint cliques. The neighbourhood property can be defined as: if there is an edge $(i,j)$ between two nodes then the nodes $i$ and $j$ are each others neighbours. If a third node $k$ is connected to both $i$ and $j$ then $k$ is a common neighbour of $(i,j)$. The presented method generates a near optimum solution. (i.e. an optimal solution is not granted.)

**Heuristic 2:**

Compute for each edge in SOG the number of common neighbours. The edge $E$ with the highest number of common neighbours is selected. The nodes of $E$ form (a part of) the first clique $C_1$. Then the graph SOG is updated. Suppose $(i,j)$ is the edge with the highest number of common neighbours in SOG then the following edges must be deleted:

1) The edges connected to node $j$.
2) The edges $(i,k)$ connected to node $i$ if there is no edge $(j,k)$, if $j < k$, or $(k,j)$, if $j > k$.

If the graph SOG is updated then the next edge $E'$ is selected. The edge $E'$ is connected to $C_1$ and has the highest number of common neighbours. The node not in $C_1$ is added to $C_1$ and SOG is updated. This process is repeated until no edges are connected to $C_1$. Then the next clique $C_2$ is generated. Clique generation is repeated until no edges are left in SOG.

What happens when two edges have the same maximum number of common neighbours? Each time this situation occurs the number of edges that must be deleted is computed. The edge with the smallest number of deleted edges is selected. The algorithm used to find a minimum clique cover of graph SOG is:

```
Minimum clique cover of SOG

local edge

edge := select-best-node-combination(SOG)
while edge
    while edge
        update-clique-graph(edge,SOG)
        combine-nodes(edge)
        edge := select-best-node-connected-to-clique(SOG)
    elihw
    edge := select-best-node-combination(SOG)
elihw
```

The function `select-best-node-combination` returns the edge with the maximum number of common neighbours and the minimum number of edges deleted. The function `update-clique-graph` deletes the edges that must be deleted according to the previous given rules. The function `select-best-node-connected-to-clique` returns the edge connected to the current examined clique with the maximum number of common neighbours and the minimum number of edges deleted. The resulting clique cover of SOG given in 4.3 is $\{(1,2,3),(6),(9,10)\}$. The resulting clique cover of DOG is

*Clique cover of data operation graph.*
4.3.3 Combining module descriptions.

During the data operation allocation module descriptions are combined. Suppose the module descriptions of X and Y are combined into one new description Z. Then the sets of X and Y are added together:

\[
\begin{align*}
OS(Z) &= OS(X) + OS(Y) \\
TS(Z) &= TS(X) + TS(Y) \\
RS(Z) &= RS(X) + RS(Y) \\
ISO(Z) &= ISO(X) + ISO(Y) \\
ISI(Z) &= ISI(X) + ISI(Y)
\end{align*}
\]

The module library is searched through for a library module that can perform the operation set \(OS(Z)\) and the cost of that module is minimal. The multiplex cost of mod Z is computed with equation (4.3).

To minimise the number of interconnection nets the in sets \(ISO(Z)\) and \(ISI(Z)\) are changed such that the number of different nets in \(ISO(Z)\) and \(ISI(Z)\) is as small as possible. Therefore the operation set \(OS(Z)\) is divided into two sub sets called commutative operation set \(COS\) and non commutative operation set \(NCOS\). The operands belonging to the operations of \(COS\) can be exchanged to reduce the number of interconnection nets.

First the operands of the \(NCOS\) operations are added to \(ISO(Z)\) and \(ISI(Z)\). Then the operands of \(COS\) are added to \(ISO(Z)\) and \(ISI(Z)\) such that the number of interconnection nets is minimised. A simple example may be illustrative.

\[
\begin{align*}
OS(Z) &= (\,+\,+\,*\,\,*\,\,*) \\
ISO(Z) &= (R_1\ R_2\ R_3\ R_4) \\
ISI(Z) &= (R_2\ R_3\ R_2\ R_1)
\end{align*}
\]

(A) \hspace{2cm} (B)

\[
\begin{align*}
OS(Z) &= (\,+\,+\,+\,\,*\,\,*) \\
ISO(Z) &= (R_2\ R_2) \\
ISI(Z) &= (R_3\ R_1)
\end{align*}
\]

(C) \hspace{2cm} (D)

Figure 4.4. Optimisation of the in sets.

In 4.4a the not optimised solution is given. In 4.4b the non commutative divide operation is examined. In 4.4c the add operation is added to \(ISO(Z)\) and \(ISI(Z)\). The operands of the add operation are exchanged to reduce the number of nets. In 4.4d the final solution is given. The number of connected nets is reduced with \(7 - 4 = 3\) and the width of the multiplexers is also reduced with \(7 - 4 = 3\).

Each time if two mods are combined above optimisation is done.

_Clique cover of data operation graph._
5. INTERCONNECTION ALLOCATION.

The last phase of the hardware allocation consists of reducing nets and multiplexers. This is done because the percentage of area used for wiring on a chip is still increasing. A rude estimate of the total area used for wiring is 40%-60%. Therefore it is desirable to reduce the number of nets. A reduction of interconnection nets can be obtained by use of so called register files. A register file is a set of registers that are grouped together. A register file has only one data input and one data output. The interconnection allocation is based on clique covering of a weighted graph to find registers that can be grouped together into one register file.

In paragraph 5.1 the characteristics of register files are given. In paragraph 5.2 the creation of the register forbid graph is declared. In 5.3 the construction of the register prefer graph is given. In paragraph 5.4 the clique covering algorithm is detailed. And finally in paragraph 5.5 an example of the use of register files is presented.

5.1 DESCRIPTION OF A REGISTER FILE.

A said before a register file is a set of registers that are grouped into one storage module. A register file consists of at least two registers. During a timeslot one register of the file can be used for retrieving data and one register can be used for writing data. To select a register for reading data a control port, called read-select is, needed. Also there must be a control port, called write-select, to select a register for writing data in the register file. The width of the control ports depends on the number of registers in the file. If there are \( R \) registers in a file then the width of the control ports is equal to the floor of \( \log_2 R \).

![Figure 5.1. Example of a register file.](image)

When the inputs of a register file steams from different sources an additional input multiplexer is required for proper selection of a source. It is clear that the total area of a register file is greater than the implementation with separated registers. This is due to the area used for implementing the control of the register file. But interconnection area is reduced remarkably and therefore the total chip area is reduced. Also the area needed for control is small compared to the total area and besides that the relative amount of it

*Description of a register file.*
decreases when the number of registers in a register file increases.

5.1.1 When is use of register file valuable?

Nevertheless register files reduce the total used chip area, not every random grouping results in a smaller amount of area. Only if the following two conditions are satisfied the registers $R_i$ and $R_j$ can be grouped successfully.

1) The registers $R_i$ and $R_j$ are never used simultaneous in a timeslot $T_k$, $0 \leq k < T$, for reading or writing of data.

2) The registers $R_i$ and $R_j$ read from the same source (or write to the same destination) at least once in at least one timeslot.

If condition 1 is not valid a register file can not be used because only one register can deliver data or store data in a timeslot. If condition 2 is not valid then the number of connections is not reduced and even can be increased. This situation is depicted in figure 5.2.

![Figure 5.2](image)

**Figure 5.2.** Non common sources or destinations can lead to increase of area.

In figure 5.2 the multiplexers required for proper selection of an input of a module are not depicted. Also the control nets for the proper selection of the operation performed by a module are not depicted. The dashed nets are nets coming from other modules.

5.2 CREATION OF THE REGISTER FORBID GRAPH.

To determine registers that can be combined into one register file a graph is constructed. This graph is called register forbid graph $RFG(V,E)$ and consists of the node set $V$, representing the registers, and the edge set $E$ representing forbidden combinations of registers. A combination of registers $R_i$ and $R_j$ is forbidden if $R_i$ and $R_j$ are used simultaneous as input or as output register in a timeslot $T_k$, $0 \leq k < T$.

If there is an edge in $RFG$ between $R_i$ and $R_j$ then $R_i$ and $R_j$ may not be combined into one register file. Condition 1, given in the previous paragraph, is then valid by construction.

*Creation of the register forbid graph.*
The graph RFG is constructed as follows:

Scan through timeslots. For each timeslot $T_k$, $0 \leq k \leq T$, two lists are build, i.e. an input-forbid-list and an output-forbid-list. The input-forbid-list is a list of registers used in $T_k$ as first or second operand of an operation. If the operation is a transfer operation then the register used for a put, write or mwrite operation is added to input-forbid-list. The output-forbid-list is a list of registers used in $T_k$ as result operand of an operation. If the operation is a transfer operation then the register used for a get, read or mread operation is added to output-forbid-list.

If a timeslot $T_k$ is scanned through the graph RFG is updated. For each combination of registers $R_i$ and $R_j$ in the input-forbid-list and the output-forbid-list an edge $(i,j)$ is included in RFG if the edge $(i,j)$ is not already in RFG. The RFG construction algorithm is:

```
local k, j, input-forbid-list, output-forbid-list
for each timeslot $T_k$ in *time-slots*
  input-forbid-list := nil
  output-forbid-list := nil
for each instruction $I_j$ in $T_k$
  if operation($I_j$) is data operation
    do
      if first-operand($I_j$) is register
        input-forbid-list := first-operand($I_j$) + input-forbid-list
      if second-operand($I_j$) is register
        input-forbid-list := second-operand($I_j$) + input-forbid-list
      if result-operand($I_j$) is register
        output-forbid-list := result-operand($I_j$) + output-forbid-list
    od
  else
    do
      if operation($I_j$) is one of {get,read,mread}
        if var-used($I_j$) is register
          output-forbid-list := var-used($I_j$) + output-forbid-list
      if operation($I_j$) is one of {put,write,mwrite}
        if var-used($I_j$) is register
          input-forbid-list := var-used($I_j$) + input-forbid-list
    od
  rof
rof
```

5.3 CREATION OF REGISTER PREFER GRAPH.

The register prefer graph $RPG(V,E)$ consists of a node set $V$ representing the registers and an edge set $E$ representing node pairs. The edge set $E$ of RPG depends on the earlier constructed RFG graph. There is an edge $(i,j)$ between nodes $i$ and $j$ if and only if:

```
Creation of register prefer graph.
```
1) There is no edge between nodes i and j in RFG.

2) The registers belonging to nodes i and j have a common source or a common destination at least once.

The second restriction satisfies condition 2 given in paragraph 5.2. Each edge E in RPG has a weight. The weight of an edge \((i,j)\) depends on how many times the registers \(R_i\) and \(R_j\) have a common source or destination. The minimum weight is 1. To find out if two registers \(R_i\) and \(R_j\) have a common source or destination the *mod-vector* is scanned through. For each mod \(X\) in *mod-vector* the result set RS\((X)\), in set IS0\((X)\) and in set IS1\((X)\) are examined. First all internal variables are deleted from these sets. Lets call the resulting sets RS', IS0' and IS1' respectively.

Then for each set combinations of registers \(R_i\) and \(R_j\) are formed. If a register combination is not forbidden (see RFG) then the edge weight of \((i,j)\) is incremented with one if there is an edge in RPG or a new edge is included. After scanning through the *mod-vector* the graph RPG is constructed. The RPG construction algorithm is:

**Construction of register prefer graph RPG.**

```plaintext
local RS', IS0', IS1'
for each mod \(M_i\) in *mod-vector*
    RS' := remove-non-register(RS(M_i))
    IS0' := remove-non-register(IS0(M_i))
    IS1' := remove-non-register(IS1(M_i))
    update-register-prefer-graph(RS'(M_i, RFG))
    update-register-prefer-graph(IS0'(M_i, RFG))
    update-register-prefer-graph(IS1'(M_i, RFG))
```

In figure 5.3 a simple example is given of the register forbid graph and the register prefer graph. These graphs are the resulting graphs for the example given in paragraph 5.5.

![Diagram of register forbid graph and prefer graph](image)

**Figure 5.3.** Example of register forbid graph and register prefer graph.

5.3.1 Is the register prefer graph a chordal graph?

For the register prefer graph RPG a clique cover is computed to find a minimum number

---

8. *mod-vector* holds a description of the mods and their interconnections found in the previous allocation phases.
of register files. If the graph RPG is a chordal graph then the clique covering can be done easily. Unfortunately is the graph RPG not a chordal graph.

A counter example is used to proof this assumption. Suppose the timeslot description is:

\[ T_0 = [(\text{get } v_1 \text{ nil nil}),(\text{get } v_2 \text{ nil nil})] \]
\[ T_1 = [(\text{+ } v_3 v_1 v_2),(\text{v_6 v_1 v_2}),(\text{get } v_7 \text{ nil nil})] \]
\[ T_2 = [(\text{+ } v_5 v_3 v_4),(\text{v_8 v_6 v_7}]) \]
\[ T_3 = [(\text{put nil v_5 nil}),(\text{put nil v_8 nil})] \]

The result of the variable allocation could be: \( R_0 = \{v_3\} \), \( R_1 = \{v_1,v_5,v_7\} \), \( R_2 = \{v_2, v_6\} \), \( R_3 = \{v_4, v_8\} \).

The timeslot description after variable allocation is:

\[ T_0 = [(\text{get } R_1 \text{ nil nil}),(\text{get } R_2 \text{ nil nil})] \]
\[ T_1 = [(\text{+ } R_0 R_1 R_2),(\text{R_2 R_1 R_2}),(\text{get } R_3 \text{ nil nil}),(\text{get } R_1 \text{ nil nil})] \]
\[ T_2 = [(\text{+ } R_1 R_0 R_3),(\text{R_3 R_2 R_1}]) \]
\[ T_3 = [(\text{put nil } R_1 \text{ nil}),(\text{put nil } R_3 \text{ nil})] \]

The resulting register forbid graph is depicted in figure 5.4a. Suppose that the add operations and the subtract operations are mapped on an add module and a subtract module successively. The mod description for the add operations is:

\[ \text{RS}(\text{add}) = \{R_0,R_1\} \]
\[ \text{IS0}(\text{add}) = \{R_1,R_2\} \]
\[ \text{ISI}(\text{add}) = \{R_0,R_3\} \]

The mod description for the subtract operations is:

\[ \text{RS}(\text{add}) = \{R_2,R_3\} \]
\[ \text{IS0}(\text{add}) = \{R_1,R_2\} \]
\[ \text{ISI}(\text{add}) = \{R_1,R_2\} \]

The resulting register prefer graph RPG is depicted in figure 5.4b.

It is easy to understand that the register prefer graph is not a chordal graph, i.e. there is no chord between (0,2) and (1,3).

**5.4 CLIQUE COVER OF THE REGISTER PREFER GRAPH.**

Each edge in RPG has a weight. It is clear that it is preferable to combine registers with the highest weight first. Therefore a clique cover of the register prefer graph is computed. The algorithms used for the clique cover of the data operation graph can also be used for the clique cover of the register prefer graph. Only one slight change has to be made.
made. The edges with the highest weight must be extracted first from the register prefer graph.

Clique cover of the register prefer graph.
5.5 Example Use of Register Files.

In this paragraph a simple example is given, where the use of register files reduces the total area remarkable. The description of the input *time-slots* is given in figure 5.5.

In figure 5.5 there is only one add operation in each timeslot. In each timeslot several values must be stored. The minimum number of registers is 4. The resulting implementation for the case that no register optimisation is done is given in figure 5.6. The six add instructions are all mapped onto one module called module_0. For proper selection of the inputs of this module two input multiplexers are used. Also there are multiplexers used for proper selection of the inputs of the registers R_0, R_2 and R_3. Port_1 is used as an io port, port_2 and port_3 as an input port.

Figure 5.5. *time-slots* description of example.

Example use of register files.
Figure 5.6. Resulting implementation without use of register files.

The characteristics of this implementation are:

- Number of control nets 10.
- Number of data nets 17.
- Number of multiplexers 5, with total width 10.
- Number of registers 4.

This implementation can be improved if the previous described methods are used. The corresponding register forbid graph (RFG) and register prefer graph (RPG) are depicted in figure 5.3. The registers $R_0$ and $R_3$ are grouped together into one register file. Also register $R_1$ and $R_2$ are grouped together. The resulting implementation is depicted in figure 5.7.

Example use of register files.
Figure 5.7. Final implementation after net optimisation.

For this implementation the characteristics are:

- Number of control nets 8.
- Number of data nets 8.
- Number of multiplexers 2, with total width 6.
- Number of register files 2, with total number registers 4.

*Example use of register files.*
6. EXTENSIONS OF ALLOCATIONS.

In the previous three chapters the variable allocation, operation allocation and interconnection allocation are discussed. The variable allocation was done before the operation allocation. In this chapter the extensions and changes are given when the operation allocation is done before the variable allocation. Some minor changes for both allocations have to be made. In paragraph 6.1 the extensions of the operation allocation are given. In paragraph 6.2 the extensions of the variable allocation are explained.

6.1 EXTENSION OF THE OPERATION ALLOCATION.

If the operation allocation is done before the variable allocation then the registers used for storage of store variables are not known. But this information is used in the estimation of the mux cost of a mod (see 4.2.3). In 4.2.3 the number of different nets is known and therefore the number of inputs of the required input multiplexer. The mux cost of this input multiplexer is computed with cost function \( c_f \). If the operation allocation is done before the variable allocation a new multiplex cost estimator must be implemented. The following heuristic is used:

**Heuristic 3:**

An estimation of the multiplex cost can be found if for each variable \( v \) the \( mux-cost \) is computed. The \( mux-cost \) of \( v \) is an estimate of the probability that a variable \( v \) can be combined with another variable \( v' \) in one register.

6.1.1 Calculation of mux cost of store variables.

The \( mux-cost \) of a variable \( v \) lies between \([0, 1]\). If the \( mux-cost \) of \( v \) is 1 then \( v \) can not be combined with a variable \( v' \) in a register. On the other hand if \( mux-cost \) of \( v \) is 0 then \( v \) can always be combined with another variable \( v' \) in a register. Two remarks can be made against the \( mux-cost \) of a variable \( v \):

1) The \( mux-cost \) of \( v \) depends only on the number of variables that are live in the same timeslot(s) as variable \( v \).

2) An internal variable can not be combined with another variable. Therefore the \( mux-cost \) of an internal variable is 1.

For each store variable \( v \) a list of variables that may not be combined with \( v \) is maintained. This list is called \textit{variable-mux-forbid-list}. \textit{Variable-mux-forbid-list} contains store variables that are live at the same time as variable \( v \). It is easy to see if two store variables are live at the same time because in *time-slots* the \textit{<timeslot-var-used-list>} is maintained.

*Extension of the operation allocation.*
Computation of mux-cost of store variables.

local $i$, variable

for each timeslot $T_i$
do
  for each variable of timeslot-var-used-list($T_i$)
    update-var-mux-forbid-list(variable,timeslot-var-used-list($T_i$))
  rof
od

for each variable
  mux-cost(variable) := compute-mux-cost(variable,var-mux-forbid-list(variable))
rof

The function update-mux-forbid-list adds all variables $v'$ of timeslot-var-used-list to the variable-mux-forbid-list of variable $v$ if $v'$ is not already in variable-mux-forbid-list. The function compute-mux-cost($v$) computes the mux-cost of a variable and depends on the variable-mux-forbid-list of $v$ and the total number of store variables.

$$
mux-cost(v) = 1 - \frac{\text{number-store-variables} - (\text{length var-mux-forbid-list}(v))}{\text{number-store-variables}} \quad (6.1)
$$

6.1.2 Computation of multiplex cost.

The multiplex cost of a mod description can now be computed with use of the mux cost of variables. Suppose IS0(X) of a mod X is $\{v_1, v_2, \text{internal}_5\}$ and the mux-cost of $v_1 = 0.4$ and the mux-cost of $v_2 = 0.5$. An estimate of the number of multiplexer inputs is then $0.4 + 0.5 + 1 = 1.9$. A multiplexer with 2 inputs is required. The multiplex cost of IS0(X) is then computed with $cf_1$. In figure 6.1 an overview of the new multiplex estimator is given.

Horizontally the sum of all mux-cost of variables is depicted. Vertical an estimation of number of multiplexer inputs is used. If the estimated number of inputs is smaller then 1 then no multiplexer is required.

Figure 6.1. New multiplex cost estimation.

Extension of the operation allocation.
6.2 EXTENSION OF THE VARIABLE ALLOCATION.

If the operation allocation is done before the variable allocation extra information is available during the variable allocation. Proper use of this information optimises the variable allocation and minimises the number of interconnection nets. Before the extended variable allocation is given the variable allocation of chapter 3 is examined.

In chapter 3 the only optimisation goal was to minimise the number of registers used to store store variables. Minimising the number of registers was done with use of a left edge algorithm. If more then one interval started at the same time randomly one interval was taken first. In the extended variable allocation this randomly selecting of an interval is eliminated. The goals of the extended variable allocation are:

1) The number of registers used for storage of store variables is minimal.

2) Try to minimise the number of interconnection nets between modules.

The above goals are achieved with a modified left edge algorithm.

6.2.1 The variable prefer graph VPG.

As stated before the information of the operation allocation is used in the extended variable allocation. As the operation allocation is done the final mod descriptions are generated. Each mod X has an in set ISO(X), an in set ISI(X) and a result set RS(X). These sets are used to form connection sets. A connection set CS consists of reduced sets ISO(X), ISI(X) and RS(X) in which the internal variables are deleted.

A simple example of connection sets may be illustrative. Suppose there are 6 connection sets CS_1 till CS_6 and the store variables are v_1, v_2, v_3, v_4, v_5 and v_6. The six connection sets are depicted in figure 6.2.

\[
\begin{align*}
CS_1 &= \{v_1, v_2\} \\
CS_2 &= \{v_2, v_3\} \\
CS_3 &= \{v_1, v_2, v_3\} \\
CS_4 &= \{v_4, v_6\} \\
CS_5 &= \{v_3, v_4\} \\
CS_6 &= \{v_5\}
\end{align*}
\]

Figure 6.2. The six connection sets CS of example.

With use of the connection sets CS a graph is created. This graph is called variable prefer graph VPG(V,E). The nodes V represent variables; the edge set E consists of preferable variable pairs. There is an edge between two nodes i and j of VPG if two conditions are valid.

- The variables i and j appear in one common connection set CS_i at least once.
- The variables i and j are not both live at the end of a timeslot T_1.

Each edge in VPG has a weight. The weight of an edge is equal to the number of times a combination of nodes i and j appear in the connection sets.

*Extension of the variable allocation.*
For the connection sets given in figure 6.2 the life time analysis is given in table 6.1. The maximum number of variables that are live in one timeslot is 3. Therefore the minimum number of registers is also 3. The resulting variable prefer graph is given in figure 6.3.

![VPG graph of the example](image)

**Figure 6.3.** The VPG graph of the example.

The graph VPG can be created in $O(P)$ with $P$ the number of connection sets.

### 6.2.2 Modified left edge algorithm for variable allocation.

As stated before in chapter 3 the store variables form a variable interval graph. The variables are 'coloured' with use of the left edge algorithm. In the modified left edge algorithm the information of the variable prefer graph is used. If more than one variable is live for the first time in a timeslot then these variables are divided into two sets. The first set, called $subset1$, consists of variables $v$ for which the following conditions hold:

1) There must be an incident node $v'$ of variable $v$ in VPG at least once.

2) The incident variable $v'$ is already in R-orderlist.

All variables $v'$ for which the above conditions not hold form the second set, called $subset0$. To divide variables into $subset1$ and $subset0$ the following algorithm is used:

---

8. This is done because if no incident nodes are ordered then the variable $v$ can be ordered randomly without net increasement.

*Extension of the variable allocation.*
Divide variables into subset1 and subset0.

```plaintext
local i, variable, subset0, subset1
R-orderlist := nil
for i = [0,T]
   for each variable element of Ti
      if (and (variable in VPG)
          (incident node of variable member of R-orderlist)
          (variable not member of R-orderlist))
         then
            subset1 := variable + subset1
         else
            subset0 := variable + subset0
         fi
   rof
   R-orderlist := R-orderlist + (subset1, subset0)
rof
```

Remark: The resulting order of variables is stored into a list called \textit{R-orderlist}. In \textit{R-orderlist} for each timeslot \(T_i\), \(0 \leq i \leq T\), the subset1 and subset0 are computed. If in a timeslot \(T_i\) a subset is not used then this subset is nil.

For the variables given in table 6.1 and the variable prefer graph of figure 6.3 the subsets are:

<table>
<thead>
<tr>
<th>(T_i)</th>
<th>subset1</th>
<th>subset0</th>
</tr>
</thead>
<tbody>
<tr>
<td>(T_1)</td>
<td>nil</td>
<td>{(v_2, v_3, v_6}}</td>
</tr>
<tr>
<td>(T_2)</td>
<td>{(v_4}}</td>
<td>nil</td>
</tr>
<tr>
<td>(T_3)</td>
<td>{(v_1}}</td>
<td>{(v_5)}</td>
</tr>
</tbody>
</table>

The modified left edge algorithm uses the generated subsets to optimise the variable allocation. The timeslots are scanned through. First the variables of subset1 are allocated. Thereafter the variables of subset0 are allocated.

Allocation of subset1 variables.

For each node \(i\) of subset1 all the edges \((i,j)\) are selected from the variable prefer graph. There is at least one such edge else \(i\) can not be in subset1. These edges form a list. This list is decreasingly ordered on the edge weight. So the edge with the highest weight appears at the front of the list.

Then the sorted edge list is scanned through. For each edge \((i,j)\) the following conditions are tested.

1) The variable \(v_j\) is already allocated into register \(R_k\).
2) The variable \(v_i\) is not yet allocated.
3) The variable \(v_i\) is not live in the same timeslot as variables in \(R_k\).

If these three conditions are valid then variable \(v_i\) is added to the register \(R_k\).

Remarks:

\textit{Extension of the variable allocation.}
- The list of edges is sorted because if two variables \( i \) and \( j \) are both incident to node \( k \) then \( i \) is placed before \( j \) in list if the weight of \((i,k)\) is higher then \((j,k)\). The combination of \( i \) and \( k \) reduces the number of connections more than the combination of \((j,k)\). If the weight is equal for both edges then randomly one variable is first added to the list.

- Not only the edge with the highest weight appears in the edge list. A simple example shows why. Suppose there are 2 variables \( x \) and \( y \) and they are both incident to variable \( z \). Let the edge weight of \((x,z)\) higher then \((y,z)\). Then \( x \) and \( z \) are combined. Further suppose \( y \) is also incident to \( w \) and the edge weight of \((y,z)\) is higher then \((y,w)\). If only the highest edge weight of \( y \) was used the combination of \( w \) and \( y \) was not detected and \( y \) was randomly grouped.

- It could be that a variable \( v \) with low weight in VPG can not be grouped because all groups of incident nodes in VPG are live in the same timeslot as \( v \). If this occurs then \( v \) is deleted from subset1 and added to subset0.

**Allocation of subset0 variables.**

Variables \( v' \) can be 'randomly' added to a register \( R_k \) for which the variables of \( R_k \) are not live at the same time as \( v' \).

The modified left edge algorithm used for the extended variable allocation is:

```
Modified left edge algorithm.

local set, variable, sorted-list, edge

for each set of R-orderlist
    for each variable of subset1
        do
            sorted-list := sort-edges-of-incident-nodes(subset1)
            for each edge \((i,j)\) in sorted-list
                if (and \( j \) is grouped in \( G_k \)
                    \( i \) can be grouped in \( G_k \)
                    \( i \) not already grouped))
                    then
                        add-variable-to-group(i,\( G_k \))
            od
    rof
for each variable of subset0
    add-to-free-group(variable)
rof
```

6.2.3 **Variable allocation versus extended variable allocation.**

In this sub paragraph the variable allocation of chapter 3 is compared with the extended variable allocation given in this chapter. If the variable allocation is done for the variables given in table 6.1 then the resulting registers are formed.

*Extension of the variable allocation.*
If the extended variable allocation is used then the resulting registers are:

\[ R'_0 = \{v_1, v_2\} \]
\[ R'_1 = \{v_3, v_4\} \]
\[ R'_2 = \{v_5, v_6\} \]

The connection sets given in figure 6.2 can be seen as interconnection nets. If the variable are replaced with the registers used for storage of the variables and the number of different nets in all connection sets are added together, then for the variable allocation of chapter 3 there are 11 different nets. For the extended variable allocation there are 9 different nets. Therefore the number of nets is decreased if the extended variable allocation is used (after the operation allocation).
7. POST PROCESSING THE ALLOCATION RESULTS.

In this chapter the post processors used after the previous described allocation phases are declared. Post processing must be done to achieve data structures suitable for further assimilation in the silicon compiler. In paragraph 7.1 the structure and computation of the state machine is given. In paragraph 7.2 the syntax of the datapath is given.

7.1 STATE MACHINE EXTRACTION.

The allocation of data processing nodes of the demand graph is extensive declared in the previous chapters. The allocation of control nodes is given in this paragraph. All the control nodes together form a description of a state machine.

In general two mayor types of state machines can be distinguished. The first type is a Moore machine. The characteristics of a Moore machine are that the outputs in a state only depend on the current state. The second type is a Mealy machine. The outputs of a Mealy machine depend on the current state and the current inputs in that state. In the silicon compiler a Mealy machine is used for the state machine. An advantage of the Mealy machine is that the number of different states is less then for a Moore machine.

7.1.1 Description of the state machine before allocation.

During the scheduling of the demand graph nodes (see 2.1) the incomplete initial structure of the state machine is created. The assignment of demand graph nodes to timeslots is done. Also the assignment of timeslots to states is done during the scheduling. The syntax, in SBNF notation, of the initial structure of the state machine is:

```
<state-machine>
 ::= "(" {<state>}+ ")".
<state>
 ::= "(" <current-state> "," <state-transition> ")."
<current-state>
 ::= <symbol-name>.
<state-transition>
 ::= "(" <state-transition-inputs> <state-transition-outputs> <state-transition-next-state> <state-transition-timeslot-name> ")".
<state-transition-inputs>
 ::= "(" {<input-pair>} ")".
<state-transition-outputs>
 ::= "NIL".
<state-transition-next-state>
 ::= <symbol-name>.
<state-transition-timeslot-name>
 ::= "TimeSlot-"<integer> <symbol-name>.
<input-pair>
 ::= "(" <variable-name> "," <net-vector> ")".
<variable-name>
 ::= <symbol-name>.
<net-vector>
 ::= "[" {
<value>
 ::= "0" | "1".
```

A state consists of a current state name and a state transition. A state transition consists of four fields. The first field is a list of inputs used in the current state. Each input consists of a variable name and the associated value. The second field is a list of outputs. The initial value is nil. The third field is the next state. And the fourth field is the timeslot.

*State machine extraction.*
belonging to the current state.

To fulfill the description of the state machine the following tasks must be done after the allocation:

1) Replacement of symbolic variable names in the state transition inputs with the actual net names used in the implementation.

2) Filling the state transition outputs field in each state transition.

For the second task the timeslot associated with the current state is used. For each instruction scheduled in a timeslot the proper control nets and values are generated. In figure 7.1 an overview of the different kinds of control nets is given.

Figure 7.1. Overview of control nets.

The ellipses are used for dividing the different control nets. The actual control nets are given in boxes. The different kinds of control nets are shortly described.

*mux control* is a control net for the multiplexer used at the input of a module. A multiplexer is used each time more than one net is connected to the input of a module. A module could be a pad, operation module, a memory a register module or a register file module. The value of the control net determines the selected input of the multiplexer.

*index-mux control* is a control net for a multiplexer used for proper selection of the index of a memory. The value of the control net determines the selected address in the memory.

*to-pad control* is a control net for a pad driver. A pad can be used for both input and output. For control of the data flow a control net is necessary. The value of the control net determines the direction of the data flow.

*module control* is a control net for a data operation module. A module control net is only necessary if a module can perform more than one operation. The value of the control net determines the selected operation.

*register control* is a control net for a register module. A register module could be a 'normal' register or a register file. If a register file is used an additional input multiplexer

State machine extraction.
is used if the number of different sources is more than one. An extra control net is then required.

The width of a control net, i.e. the number of signal lines, depends on the number of different 'selections' that can be made for a module. The width is equal to the floor of $\log_2(\text{selections})$.

### 7.1.2 Description state controller after allocation.

If the two tasks described before are fulfilled the syntax of the state machine is:

```plaintext
<state-machine> ::= "[<state>]{<state>"+ ")"}.
<state> ::= [" <current-state> ", ", <state-transition> "].
<current-state> ::= "[<symbol-name>]".
<state-transition> ::= [" <state-transition-inputs> <state-transition-outputs> <state-transition-next-state> <state-transition-timeslot-name> "].
<state-transition-inputs> ::= [" {<input-pair>}+ "].
<state-transition-outputs> ::= "[<state-transition-inputs>]* "].
<state-transition-next-state> ::= [" <state-transition-outputs> "].
<state-transition-timeslot-name> ::= ["TimeSlot-" <integer> | <symbol-name>].
<input-pair> ::= [" <input-mux-control> | <io-pad-control> | <memory-control> | <module-control> | <register-control>].
<output-pair> ::= [" <net-name> ", <net-vector> "].
<net-name> ::= [" <control-net-name> ", <net-vector> "].
<control-net-name> ::= ["<value>+ "].
<value> ::= ["0" | ",1"].
```

For `<control>` the 5 types given for `<output-pair>` must be substituted.

### 7.2 DATA PATH EXTRACTION.

During the synthesis several global data structures for different kind of modules are generated. The data structures depends on the type of the module. Examples of data structures generated are the databases for constants, memory and ports (see paragraph 2.3). The information stored in the different data structures is translated into one global description. Therefore two lists are created: a *module-list* and a *net-list*. Both lists hold a complete data path description of an implementation on register transfer level. So one of them is redundant. But to facilitate the further assimilation in the silicon compiler both are generated. The *net-list* is extracted from the *module-list*.

*Data path extraction.*
7.2.1 Syntax of module list.

The *module-list* is a list containing module descriptions. For each module five fields are specified. The first field is the name of the module. This must be an unique symbol name. The second field is a list of input nets of the module. This list consists of two parts. The first part is the leftmost input of a module, the second part is the rightmost input of a module. If one of them is not specified then the proper part becomes nil. The third field is a list of output nets. The fourth field is a list of control nets. The fifth field is the name of the module in the module library. If a field is not used then this field becomes nil. With those five fields a module is complete described. The syntax of *module-list*, in SBNF notation, is:

```
<module-list> ::= "(" {<module-description>} ")".
<module-description> ::= "[" <module-name> <module-in-nets> <module-out-nets> <module-control-nets> <module-lib-name> "]".
<module-name> ::= <symbol-name>.
<module-in-nets> ::= "(" <leftmost-input> <rightmost-input> ")".
<leftmost-input> ::= <net> | "NIL".
<rightmost-input> ::= <net> | "NIL".
<module-out-nets> ::= "(" <net> ")" | "NIL".
<module-control-nets> ::= "(" {<net>} ")" | "NIL".
<module-lib-name> ::= <symbol-name>.
```

7.2.2 Syntax of net list.

The *net-list* consists of a list of net descriptions. Each net is completely described by three fields. These fields are the name of the net, this must be an unique symbol name; a list of input modules of a net; and a list of output modules of a net. For each net these three fields must be specified. The SBNF syntax of the net list is:

```
<net-list> ::= "(" {<net-description>} ")".
<net-description> ::= "[" <net-name> <net-in-modules> <net-out-modules> "]".
<net-name> ::= <symbol-name>.
<net-in-modules> ::= "(" {<module-name>} ")".
<net-out-modules> ::= "(" {<module-name>} ")".
```

Data path extraction.
8. CONCLUSIONS.

- Variable allocation can be done fast and efficient when the variable allocation is translated into a minimum colouring of a variable interval graph. Minimum colouring is based on the left edge algorithm.

- Extension of the cost of combining mods with multiplex costs of required input multiplexers reduces the total area.

- Use of register files can reduce the number of interconnection nets between modules and the total number of multiplexer inputs.

- Operation allocation before the variable allocation improves the final result if the estimate of the mux cost of a variable is the exact multiplex cost for that variable.

- A complete description of a data path and a state machine on register transfer level is generated as a result of the allocation phase.

- Divide and conquer techniques reduces the computation time and memory requirements against the method based on the dynamic programming technique.

Suggestions for improvements:

- The estimation of the mux cost for the case that the operation allocation is done before the variable allocation can be optimised further. For instance the multiplex cost of a variable can be multiplied with a constant factor to reduce or increase the mux cost of a variable.

- More complex estimate functions for the combination of mods, in the current implementation the cost function \( cf_1 \) depends only on the delay and the area. Other user defined properties can be added to the cost function \( cf_1 \).
REFERENCES


References.
APPENDIX 0: Some words about the implementation.

In this appendix some aspects of the implementation are given. The relations between the files, the structure of the files and the contents is discussed shortly. The appendices in which the mayor global data structures in SBNF are described are given. The algorithms given in the previous chapters are all written in HotLisp, a dialect of CommonLisp\(^1\).

The structure of a file of the hardware allocation is divided into 6 mayor sections. Some sections are optional. The first section is the header-section. In this section some characteristics of the file are given, i.e. function, global lists, bugs, date, author etc. The second section consists of definitions of global variables. The initial value is also specified. The third section consists of a description in SBNF of the mayor data structures. The fourth section consists of macros accessing the data structure. The fifth section consists of function definitions. The sixth section consists of the names of the mayor functions declared in current file.

The syntax of each file, in SBNF notation is:

\[
<\text{file-description}> ::= <\text{header-description}> [ \{ <\text{global-variable-declaration}> \}] \\
[\{ <\text{data-structure-declaration}> \}] \\
[\{ <\text{macro-declaration}> \}] \\
[\{ <\text{function-declaration}> \}]
\]

\[
<\text{header}> ::= <\text{function-description}> \{ <\text{inputs}> \} \{ <\text{outputs}> \} \\
<\text{author}> <\text{last-access}> <\text{status}> <\text{bugs}>.
\]

The files are divided according to the chapter in which algorithms are presented. At the end files not directly related to one of the chapters are given. The prefix of all files is /users/wilgert/lisp/ on eutes4. The file /users/wilgert/lisp/lib/libaccess is not declared and contains descriptions of access functions of the module library. See for more information [Kaiser87].

Pre processing the input of the hardware allocation.

Two files are used for pre processing the input of the hardware allocation. These files are timeslot and pre-proc. In timeslot the data structure and macros for the global variable *time-slots* is declared. In pre-proc the pre-process functions are declared.

timeslot

- Cross-reference Include
  - pre-proc
  - alloc-var
  - alloc-op
  - alloc-net
  - ext-al-var

\(^1\) CommonLisp is developed on several large industries and universities in the U.S.
network

- **Function:**
  Describes macros for accessing global variable *time-slots*.

- **Global variables:**
  *time-slots*: holds description of scheduled operations.

- **Appendix:**
  See for syntax diagram of *time-slots* appendix 1.

pre-proc

- **Cross-reference Include**
  - io-alloc
  - lib/libaccess
  - array
  - constant
  - port
  - trans

- **Function:**
  Takes care of all pre processing tasks described in chapter 2.

- **Mayor functions:**
  pre-processor: starts all pre-processor tasks.

- **Global variables:**
  *constant-list*: database holding description of used constants.
  *array-list*: database holding description of used memories.
  *port-list*: database holding description of used ports.
  *variables-to-store-list*: list of variables that must be stored.

- **Appendix:**
  See appendix 2 for syntax diagrams of *constant-list*, *port-list* and *array-list*.

constant

- **Cross-reference Include**
  - pre-proc
  - network

- **Function:**
  Declaration of macros accessing *constant-list*.

- **Global variables:**
  *constant-list*: database for constants.

- **Appendix:**
  See appendix 2 for syntax diagram of *constant-list*.

array

- **Cross-reference Include**
  - pre-proc
  - lib/libaccess
network       bin/wsetmac

- **Function:**
  Declaration of *array-list*.
  Declaration access functions and update functions of *array-list*.

- **Global variables:**
  *array-list*: holds description of on-board arrays and of off-board arrays.

- **Appendix:**
  See appendix 2 for description of syntax of *array-list*.

**port**

- **Cross-reference**
  Include
  pre-proc       lib/access
  network       bin/wsetman

- **Function:**
  Describes the macros and functions for implementation of the *port-list*. Contains description of function for selecting pad drivers.

- **Mayor functions:**
  select-pad-drivers.

- **Global variables:**
  *port-list*: database containing information of used ports.

- **Appendix:**
  See Appendix 2 for overview syntax diagram of *port-list*.

**wsetman**

- **Cross-reference**
  Include
  port
  alloc-op
  est-cost
  mux-est
  network

- **Function:**
  Contains functions for computation of set manipulations.

- **Mayor-functions:**
  set-intersection: computes intersection of two sets.
  set-symmetric-difference: computes exor of two sets.
  set-difference: computes the difference of two sets.
  set-union: computes the union of two sets.
  set-equal: returns t if two sets are equal.
  set-remove-duplicates: removes duplicates of set.
  set-intersection-p: returns t if there is a common element in two sets.

**Variable allocation.**
The implementation of variable allocation is described in alloc-var.

alloc-var

- Cross-reference Include
  - timeslot

- Function:
  Variable allocation. (see chapter 3).

- Mayor functions:
  alloc-var: allocates the store variables into a minimum number of registers.
  update-net-name-variables: sets net property of variables used as store variables.
  update-time-slots-var: replaces the names of the store variables into the used register names.

- Global variables:
  *group-list*: contains description of variables grouped into one register.

- Appendix:
  See appendix 3 for an overview of the mayor data structures.

Operation allocation.

The allocation of operations into modules is described in alloc-op.

alloc-op

- Cross-reference Include
  - timeslot
    - lib/libaccess
    - est-cost
    - cli-search
    - bin/wsetman

- Function:
  Describes the allocation of data operations into library modules. Uses a heuristic clique search technique for allocation. (see chapter 4).

- Mayor functions:
  create-operator-vector: creates description of all data operations.
  create-cost-combined-modules: creates data operation graph.
  allocate-operations: allocates the data operations. *operator-vector*, *module-vector* and *cost-combined-modules* must be computed.

- Global variables:
  *operator-vector*: holds description of all data operations.
  *module-vector*: holds description of all mods.
  *cost-combined-modules*: holds estimation of costs of combined mods.

- Appendix:
  See appendix 4 for overview of *operator-vector* and *cost-combined-modules*.

cli-search
Function: Computes clique cover of a graph.

Mayor functions: group-nodes: computes clique covering.


Appendix: See appendix 5 for overview of main data structures.

est-cost

Function: Contains description of estimate functions $cf_1$ and $cf_2$. Most functions used for $cf_1$ and $cf_2$ are twice declared; the function execute-adjust-vector selects the proper functions.

Mayor functions: estimate-cost-module: implements cost function $cf_1$. estimate-cost: implements cost function $cf_2$.

mux-est

Function: Computes the additional multiplex cost of a mod. Initialise the multiplex costs.

Mayor functions: init-multiplex-cost: computes costs of multiplexers and extracts library names of multiplexers.

Global variables: *mux-cost-vector*: holds costs and library names of multiplexers.

Interconnection allocation.
Implementation of interconnection allocation is described in alloc-net.

**alloc-net**

- **Cross-reference Include**
  - timeslot
  - lib/libaccess
  - cli-search

- **Function:**
  Implements the allocation of interconnections. Uses a heuristic clique search algorithm. (see chapter 5).

- **Mayor functions:**
  - create-bus-prefer-graph: creates graph structure with register as nodes and edges between registers preferable to combine. Each edge a weight.
  - group-registers: divides the registers into register files.

- **Global variables:**

- **Appendix:**
  See appendix 6 for overview of syntax of *register-bank-vector*.

**Extension of variable and operation allocation.**

The changes made for the operation allocation can be found in mux-est. The changes made for the variable allocation is given in ext-al-var.

**ext-al-var**

- **Cross-reference Include**
  - timeslot
  - lib/libaccess

- **Function:**
  Describes the extension made for the allocation of variables if the allocation of operations is done first. (see chapter 6)

- **Mayor functions:**
  - create-variables-prefer-graph: creates weighted graph of preferable combinations of variables.
  - create-interval-group-list: computes the grouping of variables into registers.
  - update-module-vector: replaces the store variable names with the used register names.
  - update-operator-vector: replaces the store variable names with the used register names.

- **Global variables:**
  - *variables-prefer-graph*: holds description of variables prefer graph.
  - *group-vector*: holds description of variables grouped together into one register.
Post-processing the results of the allocations.

Two post-processing tasks are done. The extraction of the *module-list* and *net-list* is described in file network. The allocation of the *state-machine* is given in state-con.

network

- **Cross-reference**
  - Include
    - timeslot
    - array
    - port
    - constant
    - mux-est
    - lib/libaccess
    - bin/wsetman

- **Function:**
  Creates description of circuit on register transfer level. Generates two global variables *module-list* and *net-list*.

- **Mayor functions:**
  create-module-list: creates *module-list*, for each module the name, incoming nets, outgoing nets, control nets and libname is specified.
  create-net-list: creates *net-list*, for each net the name, incoming modules and outgoing modules are specified.

- **Global variables:**
  *module-list*
  *net-list*

- **Appendix:**
  See appendix 8 for overview of syntax of *module-list* and *net-list*.

state-con

- **Cross-reference**
  - Include
    - timeslot
    - /lib/libaccess
    - /bin/wsetman
    - port
    - array
    - mux-est
    - alloc-net
    - network
    - /users/leon/idtoic/new/controller
    - /users/leon/idtoic/src/dmg-interf
- **Function:**
  Completes the description of the *state-machine*.

- **Mayor functions:**
  create-operator-mod-mux-vector: initialise *operator-mod-mux-vector*.
  walk-through-state-machine : completes *state-machine* description.

- **Global variables:**
  *operator-mod-mux-vector*: cross reference between operations and modules and the additional multiplexers.
  *state-machine*: description of the state machine.

- **Appendix:**
  See appendix 9 for overview of *state-machine* syntax.

**Micellaneous files.**

Some files are not direct related to one of the chapter but for proper computation these files are necessary. The file adjust determines how the hardware synthesis must be done. The file display contains functions used for showing some results during the allocations. The file hard-alloc contains a description of files that must be loaded.
APPENDIX 1: Overview *time-slots* syntax.

time-slots:

{timeslot-name}

{timeslot-operation-count}

{timeslot-var-used-list}

{timeslot-non-transfer-count}

{operation-instruction}

{get-instruction}

{put-instruction}

{memory-instruction}

{result-operand}

{second-operand}

{variable}

{symbol}

{number}

{integer}

{instruction-vector}

{instruction}
get-instruction:

\[
\rightarrow \text{GET} \rightarrow \text{variable} \rightarrow \text{nil} \rightarrow \text{nil} \rightarrow \text{1}
\]

put-instruction:

\[
\rightarrow \text{PUT} \rightarrow \text{nil} \rightarrow \text{variable} \rightarrow \text{nil} \rightarrow \text{1}
\]

memory-instruction:

\[
\rightarrow \text{memory-operation} \rightarrow \text{memory-name} \rightarrow \text{index-name} \rightarrow \text{data-name} \rightarrow \text{1}
\]

operation:

\[
\rightarrow \text{symbol} \rightarrow \text{variable}
\]

result-operand:

\[
\rightarrow \text{variable}
\]

first-operand:

\[
\rightarrow \text{variable} \rightarrow \text{constant}
\]

second-operand:

\[
\rightarrow \text{variable} \rightarrow \text{constant} \rightarrow \text{nil}
\]

memory-operation:

\[
\rightarrow \text{READ} \rightarrow \text{MREAD} \rightarrow \text{WRITE} \rightarrow \text{MWRITE}
\]

index-name:

\[
\rightarrow \text{variable} \rightarrow \text{constant}
\]

data-name:

\[
\rightarrow \text{variable} \rightarrow \text{constant}
\]

contant:

\[
\rightarrow \text{integer}
\]
APPENDIX 2: Overview constant-list, port-list and array-list syntax.

constant-list :

one-constant :

constant-module-number :

net :

value :

port-list :

port-description :

port-name :

port-vector :

input-list :

output-list :

input-mux-name :

register-name :

internal-var-name :

port-name :

-out :
APPENDIX 3: Overview *group-list* syntax.

\[
\text{group-list :} \\
( \rightarrow \text{group-description} \rightarrow ) \\
\]

\[
\text{group-description :} \\
( \rightarrow \text{store-variable} \rightarrow ) \\
\]

\[
\text{store-variable :} \\
\rightarrow \text{symbol} \\
\]

\[
\text{group-vector :} \\
[ \rightarrow \text{group-description} \rightarrow ] \\
\]
APPENDIX 4: Overview of *operator-vector* syntax.

operator-vector:

mod-description:

operation-set -> result-set

in-set0 -> in-set1 -> timeslot-set

operation-number-set

library-module-list

multiplex-cost
cost-module

library-module-name:

symbol

timeslot-set:

operation-number-set:

null

operation-set:

result-set:

in-set0:

in-set1:

timeslot-set:

operation-number-set:

library-module-list:

library-module-name:

cost-module:

operation:

symbol
data-operation

integer
timeslot-number : 
\[ \rightarrow \text{integer} \rightarrow \]

operation-number : 
\[ \rightarrow \text{integer} \rightarrow \]

cost-combined-modules :
\[ \rightarrow \text{cost-row} \rightarrow \]

\[ \rightarrow \text{cost-row} \rightarrow \]

\[ \rightarrow \text{combine-mod-cost} \rightarrow \]

\[ \rightarrow \text{combine-mod-cost} \rightarrow \]

\[ \rightarrow \text{infinite} \rightarrow \]

\[ \rightarrow \text{max-cost} \rightarrow \]

\[ \rightarrow \text{tseng-cost} \rightarrow \]

tseng-cost :
\[ \rightarrow 0 \rightarrow \]
\[ \rightarrow 1 \rightarrow \]
\[ \rightarrow 2 \rightarrow \]
\[ \rightarrow 3 \rightarrow \]
\[ \rightarrow 4 \rightarrow \]
\[ \rightarrow 5 \rightarrow \]
\[ \rightarrow 6 \rightarrow \]
\[ \rightarrow 7 \rightarrow \]
APPENDIX 5: Overview *clique-graph* syntax.

**clique-graph** :

![Diagram of clique-graph]

**node-description** :

![Diagram of node-description]

**bigger-connected-nodes** :

![Diagram of bigger-connected-nodes]

**smaller-connected-nodes** :

![Diagram of smaller-connected-nodes]

**common-neighbours** :

![Diagram of common-neighbours]

**common-neighbours-edge** :

![Diagram of common-neighbours-edge]

**clique-set** :

![Diagram of clique-set]

**node-number** :

![Diagram of node-number]
APPENDIX 6: Overview \*register-bank-vector\* syntax.

\[\text{register-bank-vector} :\]

\[\text{register-description} : \text{register-forbid-list} \quad \text{register-forbid-list} : \]

\[\text{register-combine-list} : \text{register} \quad \text{register} : \]

\[\text{bus-prefer-graph} : \quad \text{bus-prefer-edge} : \]

\[\text{edge} : \quad \text{weight} : \quad \text{node} : \]

\[\text{weight} : \quad \text{node} : \]

\[\text{integer} \rightarrow \]

\[\text{integer} \rightarrow \]
APPENDIX 7: Overview extension of variable allocation syntax.

```
var-prefer-graph :

node-description :

first-node :

edge-list :

describe a first-node followed by a list of edges.

edge :

second-node : weight :

show a second node followed by a weight.
```

The diagram shows how these components are structured in a variable allocation syntax.
APPENDIX 8: Overview module-list syntax.

module-list :

module-name :

operator-module-name :

register-module-name :

multiplex-module-name :

controller-name :

array-module-name :

port-module-name :

symbol
module-control-name:

```
module-name ~ ctrl → symbol
```

module-lib-name:

```
module-lib-name
```

net-list:

```
( net-description )
```

net-description:

```
| net-name | net-in-module | net-out-modules |
```

net-name:

```
symbol
```

net-in-module:

```
symbol
```

net-out-modules:

```
symbol
```
APPENDIX 9: Overview state-machine syntax.

state-machine :

state :

current-state :

state-transition :

state-transition-inputs :

state-transition-next-state :

state-transition-timeslot-name :

state-transition-outputs :

input-pair :

output-pair :

net-name :
output-pair:

- input-mux-control
- io-pad-control
- memory-control
- module-control
- register-control

control:

( control-net-name . net-vector )

variable-name:

symbol-name

net-vector:

1 value 1

value:

0

1