Automatic generation of a processor design in IDaSS from an instruction set description in SimDes

Rietkerken, B.A.M.

Award date:
1996

Technische Universiteit Eindhoven

Faculty of Electrical Engineering
Section of Digital Information Systems

Master's Thesis:

Automatic generation of a processor design in IDaSS from an instruction set description in SimDes

B.A.M. Rietkerken

Coach : Ir. W.J. Withagen
Supervisor : Prof. Ir. M.P.J. Stevens
Period : January - August 1994

Summary

To make the development of faster and complexer computers possible, it is necessary to use a well structured method to design the computer's processor. The usual way to test the performance of the processor design during the several stages in the design process, is simulation. Simulation tools are developed at each design level. In the Processor Architecture Simulation System (PASS), developed by researchers at the Eindhoven University of Technology there are three levels. The simulator used at level 1 is a processor specific simulator generated by SimGen from an instruction set description in SimDes. At level 2 the Interactive Design and Simulation System (IDaSS) can be used.
To make automatic generation of a level 2 design from a level 1 description possible, a tool had to be written that translates a SimDes file into an IDaSS file. Apart from the fact that IDaSS works at a higher level, the main difference is that the IDaSS user interface is graphical, while SimDes does not include any graphical information at all. This information will have to be added to the design by the translation tool automatically. A method to generate this graphical information is to use a standard layout model to map the SimDes resources on. The building blocks are divided into groups and have their own global place in the IDaSS schematic. The exact placement depends on the number of resources and the need for additional blocks to make the resources compatible with the internal buses.
Another difficulty of the translation is the implementation of the actual instruction set description. The complexity of the implementation step depends on the layout model that is used for the layout generation. In the standard layout model an ALU operator and a state controller are used to take care of the instruction set execution. Some alternatives for the standard layout model have been examined. To obtain an optimal design in IDaSS an automatic data path synthesis tool will have to be developed. But in most cases a more orderly schematic will be desired. In that case another alternative for the standard layout model can be used: a distributed ALU model. With this method the ALU functions are distributed over several operators (one operator per resource).
All methods have their advantages and disadvantages. It is up to the next person who will engage in this subject, to decide which method is the most suitable.

Contents

1. Introduction

1.1 Processor design process

Who could think of a world without computers nowadays?
It all started with an immense electronic calculator, but now we are able to make computers very compact and extremely complex and fast. This makes it possible to leave more and more human work to electronical systems. These systems perform certain tasks by executing structured, concatenated instruction sequences, called programs. The design process of new computers concentrates on designing the 'brain': the processor. A processor is a logically interconnected system of algorithmic units, registers, ports and controllers. This part of the computer is important, because it fetches instructions from an external memory and uses them to perform certain operations. The performance of the complete system depends on the operating speed and the circuit complexity of this essential part of it.
During the design process choices will have to be made with respect to the processors instruction-set and architecture. These choices affect processor performance. The extend to which the processor performance is affected cannot be established easily. This information is vital however in the early steps of processor design. It might be impossible to obtain a certain performance, if an incorrect decision is made at an early state in the design. This necessitates the use of an efficient design strategy.
In the most ideal situation, during the design process no errors are made. The idealized design process leads 'directly' to the required solution. An abstract model for a general design process was analyzed in [Koom90] and is presented in figure 1. This model consists of a concatenation of several design steps. Each design step is accompanied by validation of the step made, validating the design decisions made on this level of detail. Although formal verification should be used

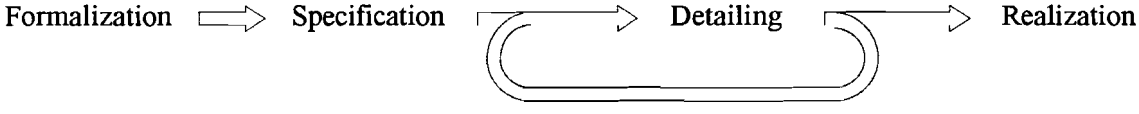

Figure 1: Idealized design proces

where possible, simulation is often the only way to validate a design step. The idealized design process consists of these four basic steps:

- Formalization step: it represents the translation of an idea into a formal description. All design-requirements are established in this step.

- Specification step: during this step, information is added to the design regarding design knowledge.

- Detailing step(s): during a detailing step extra detail is added to the result of the previous step. Subsequently the resulting problem is solved on this level of detail. Detailing is continued until a solution is found which satisfies the specification in sufficient detail to make the realization step.

- Realization step: it establishes the solution or end-goal in terms of the target system. For example, when designing an integrated circuit layout this implies generating masks for the

circuit layout.

This idealized process yields a hierarchical design; this design is described using several levels of detail. Each level of detail limits the problem-scope, freeing the design from excessive detail. Design decisions made on each design level should be validated either by formal proof or using simulation, comparing the required with the actual behaviour.

1.2 PASS: processor design tool set

Tools which establish processor performance in the design phase are rare. This was the main incentive for researchers at the Digital Information Systems group at the Eindhoven University of Technology to design PASS, the Processor Architecture Simulation System. PASS is a tool set which allows a designer to describe an instruction set processor and simulate its behaviour. Within PASS three hierarchical layers can be distinguished. These levels of progressive detail are presented in figure 2.

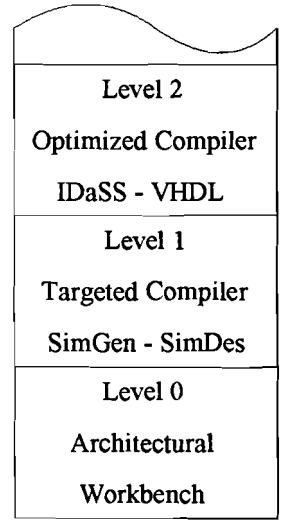

Figure 2: PASS hierarchie

The 'Architecture Workbench', [Torr88] represents level 0 in the PASS processor performance analysis system. This processor performance analysis tool was developed by the Stanford University Computer Science Laboratory. Only the basic processor characteristics are described at this level. Any alterations of processor characteristics immediately affect simulation results.
Level 1 of the PASS hierarchy is used to model the instruction set architecture. Little information regarding the actual processor structure is present at this level. At this point the instruction set processor is specified by its instruction set, and the elements upon which it operates. The PASS tools on this level generate a compiler and simulator using a processor description. This compiler-simulator pair can process benchmarks or typical applications.
Level 2 of the system hierarchy is used to model the processor architecture. It should provide a yet more detailed behavioral or even structural simulation using a hardware description language such as VHDL or IDaSS.

1.3 Interface between SimDes and Idass

In this report a disquisition is presented of the possibility to create an interface between SimDes at level 1 and IDaSS at level 2. SimDes, a processor description language, describes the processor at the instruction set level. This description is used by SimGen, the Sequential Instruction Set Simulator Generator, a tool which automatically constructs an instruction set simulator from the SimDes description [Takk92]. When the designer has finished optimizing his processor design at level 1, and simulating it with the SimGen simulator, he can start implementing it at the next higher level, using IDaSS. This Interactive Design and Simulation System describes a digital circuit using 'blocks', such as registers, ALU's, controllers and the like. It has a graphical user-interface.
At present the designer uses the design tools of the PASS hierarchy separately. Designing in IDaSS always starts with placing blocks in an empty schematic. It would save a lot of time if there would be a tool that automatically derives an IDaSS design from the SimDes description. At first it is not necessary that this tool optimizes the IDaSS design in any way. The meaning of the first part of this project consists of creating a tool which generates an IDaSS layout from the description; the implementation of the instruction set functions will be left out of consideration. The next step will be the investigation of how to implement the instruction set.
The project, described in this report, can be divided into several parts:

- Creating a tool, which automatically generates an IDaSS layout from a processor description in SimDes, without the instruction set implementation.

- Investigating the difficulties which may occur trying to implement the instruction set.

- Investigating the connection between the implementation alternatives and the layout required to support an implementation.

In the next two chapters the source (SimDes) and the target (IDaSS) are described. Information regarding the SimDes structure as well as the IDaSS storage file format is a necessity for designing the translation tool. Chapter 4 deals with the layout generation tool itself. It describes how the IDaSS blocks and interconnections can be derived from the processor description. An investigation of the instruction set implementation follows in chapter 5. Chapter 6 holds the conclusions and gives a number of recommendations for further development of this tool, not only regarding the implementation of an instruction set, but also future changes in source as well as target specifications.

2. Source description: SimDes and SimGen

2.1 Introduction to SimGen

The Sequential Instruction Set Simulator Generator (SimGen) is a tool which operates at the intermediate level of the PASS hierarchy (figure 2). It generates simulators automatically, using instruction set processor descriptions. A simulator, generated by SimGen, can be used for testing the performance of a processor at the instruction set level. It exists of a number of standard building blocks, extended with some blocks that are processor specific. SimGen will generate this processor specific parts out of the processor description. Some specifications of SimGen, applying either to the simulator generator itself, or the simulation results are:

- Simulator Generator specifications
  - Sequential processors
  - Behavioral simulation
  - Language C, machine independent
  - Instruction is the smallest executable element

- Simulation specifications
  - Read and write accesses to Memory
  - Read and write accesses to registers and ports
  - Arithmetic Operations
  - Function Usage
  - Instruction Usage

The processor model used by SimGen simulators is similar to that used in [Miya84]. Although instructions to the outside world appear as the most elementary simulator actions, inside the processor a single instruction is divided into 4 parts, see figure 3. These parts are reminiscent of the 'Von Neumann' stages. The fetch, the decode and the execute stage represent instruction execution.

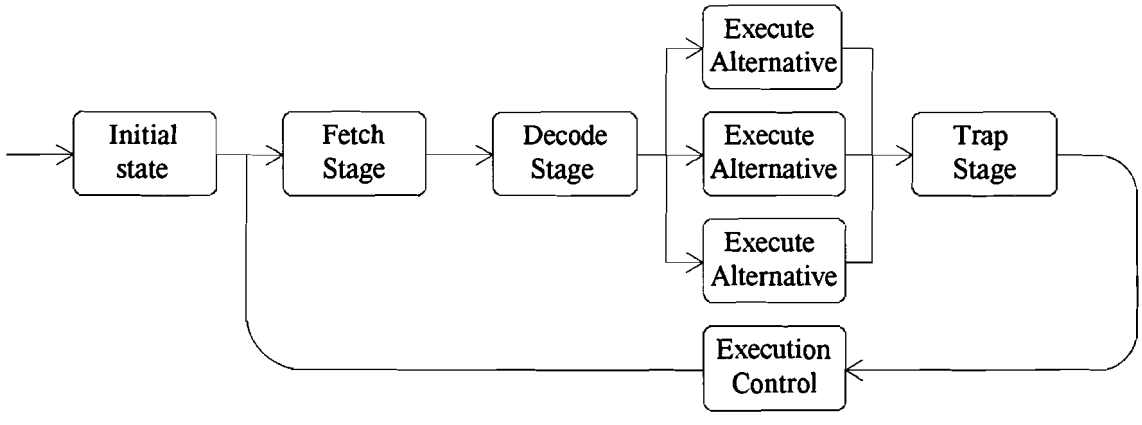

Figure 3: Instruction cycle

- Fetch stage: information needed for the instruction decoding process is retrieved from

memory. This process is identical for all instructions. It is modeled using one single description for all instructions.

- Decode stage: this process determines what kind of instruction is to be executed and invokes an appropriate execution stage. This decoder is automatically generated using instruction descriptions. Automatic decoder generation alleviates the processor description process.

- Execute stage: during this stage additional operands can be fetched. The instruction will be executed. This stage is unique for each individual instruction or group of instructions.

- Trap handler stage: this stage is executed after each instruction to test if interrupt(s) occurred. The smallest visible step in the processor is an instruction, it is not possible to interrupt the actual instruction execution. Similar to the fetch stage one single trap handler description is used for all instructions.

Processor state within a SimGen simulation is determined by various different pieces of hardware, referred to as resources. Three different resources are distinguished: memory, registers and ports. The characteristics of this resources must be described in the processor description.

2.2 SimDes: processor description language

2.2.1 Syntax and semantics

Initially the processor description language SimDes was designed for SimGen usage [Takk92]. Ideally a single processor description should suffice for all tools at the intermediate level of the PASS hierarchy. A SimDes description consists of several modules or blocks. This modular approach eases expansion of the language with new constructs, if needed in the future. Blocks not needed by one tool can be discarded by its scanner. Blocks are braced using |[ ]| brackets. The block type follows the opening bracket. All blocks have a fixed position in a processor description. Some blocks may appear more than once in a single description. The block types used by SimDes are:

aliases	instructions
comment	memory
description	PC
fetch	ports
fetchbuffer	registers
functions	resources
history	traphandler
implementation	statistics

A designer usually compares several alternative processor implementations. The SimDes language allows the description of multiple implementations of a single processor in one single description file. Once this description is processed by SimGen, the simulator generator returns two separate simulators, one for each individual implementation. A description consists of a global part and at least one implementation block (see figure 4). In the global part all global resources and global statistics are described. These resources and statistics are accessible to all implementations. Implementation resources are accessible within the implementation in which they are declared.
For a list of all production rules see appendix B.

```
|[ description  : ...

  |[ resources  : ...  ]|                    # resources visible to
  |[ statistics : ...  ]|                    # all implementations

  |[ implementation : <impl_name>
     |[ resources  : ...  ]|                 # resources visible
     |[ statistics : ...  ]|                 # only to this
     |[ ....       : ...  ]|                 # implementation
  ]|

  |[ implementation : <imp2.name>
     |[ resources  : ...  ]|                 # resources visible
     |[ statistics : ...  ]|                 # only to this
     |[ ....       : ...  ]|                 # implementation
  ]|
]|
```

Figure 4: Description structure

2.2.2 Global description blocks

There are two blocks present within the global part of the description: the global resources block and the global statistics block. The global resources block is used to define registers, ports and aliases which are accessible to all implementations. In this block the two obligatory fields, memory and program counter, must be present otherwise SimGen can not construct a functional simulator. Although memory is not part of the processor as such, applications cannot be simulated without it.

```
|[ resources   :
   |[ memory    : ...  ]|
   |[ PC        : ...  ]|
   |[ registers : ...  ]|
   |[ ports     : ...  ]|
   |[ aliases   : ...  ]|
]|
```

Figure 5: Global resources declaration

From all these resources parameters like name, bitwidth and size of an array, can be declared. Only for the PC and the registers it is possible to assign an initial value. The PC and port declarations actually are special cases of register declarations. For the PC the speciality lies in the fact that the SimGen simulator will automatically update its value, once an instruction is decoded, in accordance with the instruction opcode length. Ports only differ in the feature that they cannot be initialized. At the moment the special nature of ports, io devices, is not modelled by SimDes. The only reason why they are declared in a separate block is because it might be useful for future extensions. Aliases may be used to redefine parts of previously declared resources, such as registers or ports, to ease the access to this specific parts.
Statistics are declared separately because they do not model processor hardware. Statistics can be used to store information which is not logged elsewhere. The processor designer will have to update this statistics by including special statistics statements in the instruction set functions. In a processor description statistics are addressed similar to registers. Statistics are restricted in their use, reflecting their true nature. It is not allowed to use statistics in alias declarations and as counters in 'for' statements.

2.2.3 Implementation blocks

An implementation description consists of various implementation specific blocks (figure 6). Implementation specific resources are declared within the implementation resource block and are accessible in this implementation only. The most important block in the implementation resources block is the fetchbuffer

```
|[ implementation  : ...
  |[ resources     :
    |[ fetchbuffer : ... ]|
    |[ registers   : ... ]|
    |[ aliases     : ... ]|
  ]|
  |[ statistics    : ... ]|
  |[ functions     : ... ]|
  |[ fetch         : ... ]|
  |[ instructions  : ... ]|
  |[ traphandler   : ... ]|
]|
```

Figure 6: Implementation structure

block. This buffer is used by the instruction decoder to recognize instructions and extract operands. The fetch block specified by the processor designer places the next instruction to be processed in the fetchbuffer (figure 7). The fetchbuffer is then used to decode the instruction in question. Other blocks in the implementation resource block correspond to those in the global declaration block.

```
|[ fetch :

   {                                # no local variables involved
                                    # assume a fetchbuffer of 3 bytes called FB
                                    # assume a memory declaration named M
                                    #
    |                               # | symbol must be present
                                    #
     FB[0] = M[PC];                 # most significant word
     FB[1] = M[PC+1];               #
     FB[2] = M[PC+2];               # least significant word
   }
]|
```

Figure 7: Fetch block example

The implementation statistics block can be seen as an extension of the global statistics specification, and has exactly the same shape and function.
One of the blocks that are particular for the implementation part of the description is the functions block. SimDes supports a function mechanism. This avoids the repeated coding of certain operations such as BCD-additions, signed additions, address calculations and the like. SimDes functions are rather basic functions, their arguments are restricted to simple types as is the optional function return element. Arguments must be declared in the function header, similar to C. If needed, local resources can be declared inside a function. Similar to C, the user may end function execution using a return statement. The return is either followed by a return argument in parenthesis or nothing at all. If no return is specified the function simply returns after reaching the end. An example function is shown in figure 8.

```
|[ functions :

   Add(op1,op2 {8}) : {8}                # function with two 8 bit
                                         # parameters

     { result {8} |                      # local variable 8 bits

       result=op1+op2;                   # Add parameters
       return(result);                   # Return result
     }
]|
```

Figure 8: Function declaration example

The instruction block is an important block in a processor description. This block contains all instruction declarations for a single processor implementation. The instruction declarations contain information regarding the actual opcode, instruction operands retrievable from the instruction opcode, and a behavioral description. The instruction description must specify the characteristic pattern or opcode for each instruction. Often small operands are contained in the opcode information. Instruction operands are retrieved automatically from the fetchbuffer and are made accessible to the instruction behaviour description. Similar to a function block, an instruction block may contain local resource declarations. In figure 9 a simple instruction is declared for a fictive processor.

```
|[ instructions :

   Add ( 128,8,0 | RegId{0..2}, dis{3..7})

     {
        Temp {8}
     |
        Temp = Register[RegId] + M[IX+dis];
        Accu = Temp;
     }
]|
```

Figure 9: Instruction block example

The instruction declarations from each implementation block are used by SimGen to generate an instruction decoder. The instruction decoder uses the opcode and operand specifications, to select an instruction from the instruction set which matches the contents of the fetchbuffer.
The traphandler block resembles the fetch block. Just like the fetch block it is executed during each instruction cycle. It consists of a declaration part for local resource declarations and a statement list.

2.2.4 Statements

Statements in SimDes are part of statement lists. Each statement in such a list can be a compound statement, a statement list in braces. In this respect it is not unlike C. However one should note that it is impossible to define new variables within a compound statement. SimDes contains 7 statement types. For a more elaborate definition of SimDes statement syntax consult appendix B.

If construct

```
if expr then stat
if expr then stat else stat
```

The `if` construct is used to control program flow. If the expression `expr` does not evaluate to 0, the expression is considered to be true. In this case the `then` clause is executed. If the expression does evaluate to 0, the expression is considered to be false. In this case the `else` clause is executed. If no else clause is specified the next instruction in the statement list is executed.

For construct

```
for designator1 = expr     to expr do stat
for designator1 = expr downto expr do stat
```

The `for` construct closely resembles the Pascal for construct. `designator1` can be any resource, including aliases. The `for` loop is executed a fixed number of times. The expressions in the for statement are evaluated at loop entry and are not influenced by assignments to variables in the loop body.

While construct

```
while expr do stat
```

As long as expression `expr` does not evaluate to 0 `stat` is executed.

Repeat construct

```
repeat stat until expr
```

Execute `stat` and continue until the guard expression `expr` evaluates to 0.

Case construct

```
case expr
     {
      const1  : stat
       ...
      default : stat
     }
```

The `case` statement is used to select one alternative from a set of alternatives. At case entry expression `expr` is evaluated and according to it's value an alternative is selected and executed. All alternatives are characterized by a constant. All constants in a single `case` statement should be unique within that statement. The `default` clause is compulsory.

Return construct

```
return
return(expr)
```

The `return` construct is used to terminate block execution, it can be used in any statement list. In a function body the `return` statement returns the function value, thus if a function is void, use

the `return` statement without argument i.e. return no value.

Assignment construct

```
var   = expr   ( direct assignment )
var  *= expr   ( multiplication    )
var  /= expr   ( division          )
var  %= expr   ( remainder         )
var  += expr   ( addition          )
var  -= expr   ( substraction      )
var <<= expr   ( left shift        )
var >>= expr   ( right shift       )
var  &= expr   ( bitwise and       )
var  ^= expr   ( bitwise xor       )
var  |= expr   ( bitwise or        )

var ++         ( increment         )
var --         ( decrement         )
```

Assignments come in many different formats. Each assignment comprises of a variable designator in the left hand-side and an expression as right-hand side. The left-hand side can be any resource. Two different operations can be distinguished, the direct assignment, and various indirect assignments. In indirect assignments, a binary expression using both left and right-hand side as operands is evaluated, before the result is assigned to the left-hand side. For example
`var *= expr` equals `var = var * expr`.

2.2.5 Expressions

Expressions can occur in various positions in a processor description. For instance as conditions, as operands in assignments and as function arguments. Expressions may be defined recursively. For the production rules, see appendix B. In expressions operators will be used with a syntax that resembles that of C. All operators are tabulated together with their precedence and associativity in tables 1 and 2.

Table 1: Operators

Arithmetic		Logical		Bitwise	
*	multiply	==	equal	&	bitwise and
/	divide	!=	not equal	^	bitwise xor
%	remainder	<	less	|	bitwise or
+	and	<=	less equal	~	bitwise negate
-	minus	>	more		
<<	left shift	>=	more equal		
>>	right shift				

Table 2: Operator characteristics

operators	associativity	precedence
{}	left	high
~	right	..
*,/,%	left	..
+,-	left	..
<<,>>	left	..
<,>,<=,>=	left	..
==,!=	left	..
&	left	..
^	left	..
|	left	low

Consider a binary expression which adds two 4 bit registers. The result of this operation may consist of 5 bits. At this point there are two options, the numbers are added, resulting in a 5 bit number which must be reduced to 4 bits before assigning it to a register, or alternatively the result can be limited to 4 bits automatically. SimDes currently supports the latter implementation.
If a number is added to a four bit register, the result will be four bits. If two registers of different types are added, the result will have a type corresponding with the largest of the registers involved. This is not the case for shift operations. In a shift operation the left-hand operand is shifted. The result type should therefore equal that of the left-hand expression operand. The complete 'type' promotion scheme is presented in table 4.
Numbers are used quite often in expressions, numbers don't have a type. When numbers are multiplied, the result should not be limited to a certain width. To distinguish numbers from other expressions, numbers have a width of 0. Other expressions in SimDes have a width of at least 1, thus once an expression of width 0 is encountered, that expression must be a number. Subsequently arrangements can be made to evaluate the expression containing only numbers accordingly. A width of 0 furthermore implies that when adding a number to a resource, the result automatically inherits the resource width instead of width 0. Thus the promotion scheme in table 4 can be used.
SimDes contains a single operator not found in C, the slice operator. The slice operator is a ternary operator, it accepts three arguments. Each argument in turn may be an arbitrary expression.

```
expr { bit_selector_low .. bit_selector_high }
```

If the bits specified within the braces do not protrude outside the expression width, then the corresponding bits are extracted from expression `expr`. If the bits specified within the braces of a slice do protrude outside the expression width, the current expression width is extended, zeroes are added on the new positions. Subsequently the selected bits are extracted and aligned to the right-hand side.
The converse of slicing is concatenation. The concatenation of two expressions yields one new expression. At this time no such operator is present. Alternatively a processor design can merge

two expressions by assigning them, one at a time, to only a part of the resource. To mask these multiple access one could assign the parts to a dummy, and subsequently assign this dummy to the actual variable.

Table 4: Type promotion and operators

expression	operator	result width (= type)
expr { expr1 .. expr2 }	slice	σ(expr2)-σ(expr1)+1
expr1 <op> expr2	arithmetic	*max*(τ(expr1),τ(expr2))
	bitwise	*max*(τ(expr1),τ(expr2))
	shift	τ(expr1)
	logical	0
<op> expr	bitwise	τ(expr)

Note: τ(x)= width of x, σ(x)= value of x

2.2.6 Language evaluation

Limitations to SimDes

The most important limitations are:

- Single memory
  Currently only a single memory declaration is allowed. A memory manager was introduced to avoid the declaration of a memory array. The current memory manager supports only a single memory declaration.

- Parallelism
  The current description language doesn't contain timing information. Timing information is less important in sequential simulations. When trying to simulate a parallel processor, e.g. a pipelined processor, the need for timing information becomes evident.

- Numbers are restricted to 32 bits
  Numbers are limited to the largest numeric available in C, the unsigned long. Although it is possible to describe a processor which uses registers exceeding 32 bits, the resulting processor description will be messy and will not reflect the actual processor. It would be far more elegant to allow larger numbers and solve problems in the simulator generator.

- Unsigned numbers only
  Values in the simulator are stored in resources. Numbers in these resources are represented as a sequence of subsequent bits. All numbers in SimDes are unsigned numbers. It is not possible

to assign a value of -2 to a register.

Future extensions

Apart from eliminating several of the mentioned limitations some other features could be added to the SimDes language to benefit the ease of processor description.

- Option to share functions
  Currently functions are local to a single implementation. This is not very efficient since many functions in different implementations will be identical. It is more efficient to introduce a global function block.

- Option to include descriptions
  A complete processor description must be specified in a single file. Adding an inclusion mechanism seems very useful. A possibility is to use the C-preprocessor (CPP) for this purpose.

- Concatenation operation
  One of the operations on expressions is expression slicing. Two expressions can already be merged using two assignments to a single variable. The introduction of a concatenation expression would simplify this procedure significantly. This alteration will not be very difficult. It requires an extension of the expression node, and the addition of small parse-tree build and verification functions, similar to those for the slice operator.

- Compounds
  Aliases are used to redefine a single part of a resource under a new name. Whilst aliases divide resources, compounds concatenate resources and redefine the concatenation under a new name. The implementation of compounds will involve a considerable amount of administrative overhead.

- Add cycle/state information automatically.
  Add this information to instructions and increment automatically.

- Add write statement
  Allow the user to log information which (s)he thinks is important, but is not logged by any of the log procedures. Alternatively it can be used to create a more selective log of information.

- Allow passing of arrays to functions

- If unsatisfactory, use another expression evaluation system.
  The current expression evaluation system uses type information. This expression evaluation system was introduced to simplify processor descriptions. If it should prove unsatisfactory, it should be replaced by a more general expression evaluation system. E.g. ordinary C expression handling.

2.3 Description of the SimGen tool

Once a processor has been described using SimDes, the SimGen tool can be used to build a processor specific simulator. Actually SimDes is a compiler, a tool which transforms one language into another language. SimDes is constructed out of several building blocks. Before generating the

SimGen file two of the building blocks have to be created by the GMD tool-set for compiler construction: SimGen.rex and SimGen.lalr . The creation of this files will be done only once as long as the description language syntax doesn't change. The rest of the building blocks are standard. Each group of blocks, a module, has its own function.

2.3.1 SimGen modules

The compiler consists of various stages during which operations are performed on the input language.Four phases can be distinguished in the SimGen compiler: the scanner, the parser in combination with the parse tree builder, the parse-tree checker and the code generator, see figure 10. The first three stages are part of the so-called compiler front-end. The compiler front-end is independent of the target language. The compiler back-end translates the information handed over

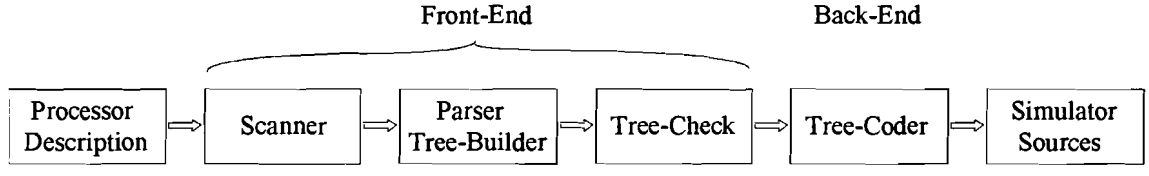

Figure 10: SimGen compiler structure

by the front-end to actual target code, in this case C source code.

SimGen consists of the following modules:

Lexical scanner

The scanner accepts a processor description, and tries to match input symbols to SimDes tokens. SimDes tokens are specified as regular expressions. These tokens are the elementary parts of the description language. If the scanner can not match the input symbols to a regular expression, this is reported to the user and the compilation process will be aborted. The SimGen scanner is constructed using rex, part of the GMD tool-set for compiler construction. Rex stands for Regular Expression tool and is the GMD equivalent of LEX a well known lexical scanner generator amongst those familiar with Unix. Rex is passed a description containing definitions of all tokens in the language. Tokens are described using regular expressions. The scanner, created by rex, matches input symbols with these regular expressions. Once a token has been recognized, a small piece of code is executed. Usually this code hands over the ScannerAttribute to the parser.

Parser

Tokens recognized by the scanner are handed over to the parser. The parser is constructed using lalr, a tool from the GMD tool-box. Lalr constructs parsers for languages based on LALR(1) grammars. A lalr parser description consists of a list of production rules and semantic actions associated with these rules. Lalr constructs the main parser module. Its input consists of the production rules presented in appendix B. The parser description is extended with so-called semantic actions. These are executed whenever a production rule is matched. The parser tries to

match sequences of tokens to the production rules. If a production rule is matched all tokens are accepted. The attributes attached to these tokens are placed in a data-structure called the parse-tree. The parse-tree and the processor description contain similar information. The processor description however is troubled with syntactic information which is not present in the parse-tree. The formal definition of the parse-tree type is presented in appendix C. For a more detailed description of the parse-tree structure, see [Takk92] paragraph 4.2 . If no production rule can be found to match the input tokens, a syntactic error has occurred in the input and will be reported. If a syntactic error occurs the compilation process will be aborted.

Parse-tree check

The parse-tree is build during the parse. After a parse has been successfully completed, the tree contains all attributes from the processor description. At this point the semantic correctness can be verified. If a semantic error occurs during this verification, the compilation process will be aborted. The parse-tree check is based on a recursive tree descent. The tree is verified one node at a time, starting with the tree root. Once a node is verified control is handed over to functions which verify all child nodes.

Code generation

During the last stage of the compilation process parse-tree information is translated to actual target code. In case of the simulator generator this implies the construction of data-structures, and the translation of behavioral descriptions to C functions.

2.3.2 How to use SimGen

In each module most of the building blocks are standard files. Some files are included in more than one building block.

All files relevant for scanner generation are:

SimGen.rex	contains regular expressions and other information used in the scanner generation process.
Token.h	contains toggles for scanner/parser log files contains token defines
ScanType.h	contains types for Scanner
ScanFunc.h	contains function prototype for assisting functions
ScanFunc.c	contains assisting functions for Scanner

Parser and tree-builder related files:

SimGen.lalr	production rules and semantic actions
Token.h	token defines and debug toggles
ScanType.h	scanner attributes and types
Treetype.h	parser attributes and types
BuildFunc.h	tree-build function prototypes
BuildFunc.c	tree-build functions

Files used in the parse-tree checker:

CheckType.h	check-type definitions and scope defines
CheckFunc.h	check function prototypes
CheckFunc.c	check function module
Check1Func.c	miscellaneous check functions
Check2Func.c	tree structure check functions
Check3Func.c	statement and expression check functions

Code-generation files:

Token.h	token defines and debug toggles
ScanType.h	scanner attributes and types
Treetype.h	parser attributes and types
CodeFunc.h	coder interface
Code1Func.c	miscellaneous coding functions
Code2Func.c	general simulator coder functions
Code3Func.c	statement and expression coding functions

A Makefile was constructed to ease the generation of SimGen. This Makefile creates a scanner using SimGen.rex and a parser using SimGen.lalr . Subsequently additional files will be compiled, and an executable named SimGen will be produced. The command for SimGen generation is:

```
make SimGen
```

SimGen is a stand-alone compiler, it can be moved to the directory where simulators should be generated.
Once SimGen has been generated and a processor description called e.g. "Test" has been written, a simulator can be generated for this file. Descriptions are handed over to SimGen using standard Unix conventions. As a result of the command

```
SimGen < Test
```

several files are generated. If logs are activated, additional information will be placed in the debug-files. If a description is incorrect, syntactically or semantically, the code-generation process will be aborted. If a description passes the syntactic and semantic verification, code will be generated for this file. The output code files are:

Define.h	constant definitions
<Implementation_name>.c	simulator main
<Implementation_name>_func.h	function prototypes
<Implementation_name>_instr.h	instruction prototypes
<Implementation_name>_var_var.h	database declaration
<Implementation_name>_var_code.h	function and instruction code

To automatically generate a simulator several standard building blocks must be present in the directory where the simulator is generated. These standard files are:

Assign.c	assignment functions
Decoder.h	decoder function prototypes
Decoder.c	decoder functions
ExprEval.h	expression evaluation prototypes
ExprEval.c	expression evaluation functions
FixVar.h	standard variable declarations

ForMacro.c	for-statement macros
MemoryMan.h	memory manager prototypes
MemoryMan.c	memory manager functions
UserIn.h	user interface prototypes
UserIn.c	user interface functions

Once these files are present in the same directory as the files obtained from the SimGen compiler, a simulator can be generated using a second makefile. For example a simulator for a "mc68k" implementation is generated by:

```
make mc68k.sim
```

This command will return a simulator by the name of "mc68k.sim". For an extensive description of the usage of the created simulator see [Takk92].

3. Target description: IDaSS

3.1 IDaSS operation

IDaSS is an interactive design and simulation environment for digital circuits [Vers90]. It is targeted towards VLSI and ULSI designs of complex data processing hardware (micro-, co-, and signal processors of all kinds). It can also be used for simpler designs, as long as the complete design is a synchronous machine (a single clock source for all clocked elements in the design). Simulating asynchronous logic with internal feedbacks is impossible with this version of IDaSS, because the built-in simulator is not designed to do so (the results will not mirror actual hardware behaviour).
IDaSS describes a design as a tree-like hierarchy of schematics. The schematics contain elements like registers, ALU's, memories, state machine controllers and the like, and are entered graphically. Rectangles (called 'blocks') represent all schematic elements, which are connected by lines representing the (bidirectional) buses. Small squares at the boundaries of the rectangles represent the input and (three-state) output ports of the elements, these are called 'connectors'. The connectors come in several shapes to make a distinction between input, (disabled) output, bidirectional and control connectors.
All IDaSS versions are implemented in a Smalltalk environment. This implies working with edit-windows and pop up menus from which items can be chosen by clicking a mouse button. After starting up IDaSS the designer can either start creating a new schematic or load an existing design from a file. By placing blocks in a schematic, assigning functions to each block and connecting the block connectors using buses, a processor can be designed. A description of the characteristics of each block follows in the next section. If a design is saved in a file, it can be used later on in IDaSS after restoring it. The storage file has a certain format, which is described in section 3.3 .
IDaSS is not only a design but also a simulation system. Especially for simulation purposes IDaSS has a 'step' function. By executing this function, IDaSS simulates what happens to the resource values within the system after going through one clock cycle. A viewer, connected to a block, connector or bus, shows its value. By examining all viewers, the designer can verify the functioning of the system. Simulation within IDaSS is immediate, there are no separate compile stages to do. Once you place an element inside a schematic, it immediately behaves like the hardware equivalent. Drawing a bus between two connectors will immediately transfer data. There is no need to stop or even reset the simulation when changes are made to the design. This is where the 'I' in IDaSS stands for - it is 'I'nteractive.

3.2 Block descriptions

Regarding the diversity of blocks which can be placed inside a schematic it is important to know what functions can be performed by each block. In this section an overview is presented of the blocks.

3.2.1 Schematic

IDaSS is hierarchical, which means that the user can place a complete schematic inside another schematic (the nesting depth is essentially unlimited). Schematics are clearly marked on the schematic drawing by having a thicker boundary. Editing a schematic is done by opening up another simulator window by choosing 'edit' from the schematic symbol menu. The new edit window is completely linked with the other windows, pushing the 'step' button will step all

schematic elements in the hierarchy.
Bidirectional connectors can be placed in a schematic to make interconnections with higher level schematics possible. The top level schematic has no symbol, and hence cannot be given connectors. If the designer wants to add connectors to it (because it must become a library element, for instance) he can file out the complete system and, after that, load it as a block within a new schematic. After adding the connectors the block contents can be filed out again. This will create a similar file as before, but now it includes information about the block symbol.

3.2.2 State machine controller

A controller can test and control the elements of the schematic it is placed in, and can change its state based upon test results. Controllers can be placed in a schematic just like all other blocks. Their operational characteristics are entered in textual form, describing a state machine.Tests done by the controller can only be based upon directly clocked elements in the schematic. Controller blocks are clearly marked on the schematic drawing by having a thick grey boundary.
Controllers are described by a numbered sequence of separate states, which each can have a label to be used for referencing during state transition statement execution. The first state executed following system reset is always state 1. Editing a controller is done by opening a special 'state editor' window, in which each state can be described in a special language. For the complete syntax of this language see [Vers90].
A state description is made up using several standard parts. The state label is a user defined name, directly followed by a colon. If a state has no label, then the colon should be present on its own. Following the colon are commands for blocks and signals, separated by semicolons. The last command may be a state transition command for the state machine itself. Conditional blocks are always placed within square brackets and may come in place of a block or signal command. These contain an expression to evaluate, and one or more conditionally executed command groups (separated by vertical bars). The expression result values under which a command group is executed may be specified at the start of each group, using constant values, ranges of values, values containing don't care bits or any combination of these.
In principle, the complete state description is evaluated, and those commands not masked out by conditional constructs are executed at the next clock step. There is one general exception: state transition commands break off description evaluation. The remainder of a state is not evaluated anymore. Several 'jump' and 'call' transitions are available.

3.2.3 Operator

Operators model all asynchronous elements (other than simple buffers - although they even could do that) in the schematic. The number of inputs and outputs is essentially unlimited, and completely user-defined. Operators can execute one or more user-defined functions. These are entered using a 'function editor' window, in which the user types a 'mathematical' description of the wanted function. The functions can range from simple bus width conversions and multiplexers to a full IEEE floating point ALU.
Which function is executed during a clock cycle is determined by the controller(s) or by a controller connector; a default can be defined by the user. A control connector selects the functions asynchronously, a change in value on the control connector's bus will immediately select another function. The control connector can also be used as an input.

3.2.4 Register

The register models exactly what its name implies: a master/slave register with 1 .. 64 bits width.

A register can have one input and/or one output. The contents of the register following system reset are user-defined. It can execute some simple operations. The default function after initialisation is 'hold', but can be changed by the user. Again a controller can determine which function must be executed by the register. Possible functions are:

'ressem'
resets the register's semaphore at the next clock. Setting the semaphore has a higher priority than resetting, so 'load', 'loadinc' and 'loaddec' set the semaphore, even when the 'ressem' command is given in the same clock period.

'reset'
loads the register with the user-defined synchronous reset contents at the next clock. This function overrules all functions given below.

'hold'
lets the register hold its present contents at the next clock.

'load'
lets the register load from its input port at the next clock.

'inc'
increments the register at the next clock.

'dec'
decrements the register at the next clock.

'loadinc'
loads the register with the incremented value present at its input port at the next clock.

'loaddec'
loads the register with the decremented value present at its input port at the next clock.

'setto: <value>'
sets the register to the given <value> at the next clock.

3.2.5 Buffer

The buffer models a simple asynchronous unidirectional three-state bus buffer. The input and output buses should have the same width. The output can only be a three-state output, which is controlled by the standard three-state output control commands.

3.2.6 Constant generator

The constant generator models a simple way to 'inject' controller generated constants into the schematic. A constant generator has one output. Combining a constant generator with a control input gives a very powerful simulation of a PLA (Programmable Logic Array). The control input has asynchronous control over the output value. A constant generator knows only one command of its own:

'setto: <value>'

lets the constant generator generate <value> at its output (enables three-state output if present). This only works for the current clock period, and is done immediately upon receiving the command (asynchronous behaviour). Only one 'setto:' command may be active at any time. A constant generator generates UNK by default, this can not be changed. If a constant generator is needed which does not generate UNK by default, a single output operator block sometimes does the trick.

3.2.7 Memory

Several kinds of memories can be included as 'on-chip-memory' in a schematic. The current size limits for all types of memory are 2048 words of 64 bits each. Memory contents can be saved to and loaded from disk. The format used is a standard Intel HEX file with default extension '.HEX'. Two types of memory, RAM and ROM are described below.

RAM

The RAM models a (multiported) random access read/write memory. Reading is done asynchronously, the output follows the address input directly. Writing is done synchronously with the clock. The number of read ports (address input plus data output) and write ports (address and data inputs) are essentially unlimited. RAMs recognise the following commands:

'write'
enables writing at all write ports at once.

'write:<dataInputName>'
enables writing at the data input port with the name <dataInputName>.

'nowrite'
disables writing at all write ports at once.

'nowrite:<dataInputName>'
disables writing from the data input port with the name <dataInputName>.

ROM

The ROM is simply a RAM without the possibility to create write ports. Read ports may not be fixed at a specific address (would not be of much use anyway). The contents do not change on system reset. The only commands given to a ROM can be commands to enable or disable its three-state outputs.

Other types of memory are: FIFO, LIFO and CAM.

3.3 IDaSS design file format

The file format used by IDaSS for storage purposes, is a textual description of the actual design, in which all design and simulation state information except for memory contents are given [Vers93]. This file follows the hierarchy of the actual design, and uses a kind of 'braces' to control the hierarchy levelling. Because IDaSS is a design and simulation system, design and simulation state information are merged in the design file. Also information needed only for

drawing purposes (schematics) have to be saved. Furthermore, comments and miscellaneous information are stored in the design file.
The IDaSS design (.DES) file format is a text file, containing ASCII characters. No control characters other than line terminators and (optionally) a file terminator are used. The current version of IDaSS will read files with lines terminated by a <LF> or a <CR><LF> pair. It will write files with the <CR><LF> pair as line terminator. Before each <CR><LF> pair a line must contain at least one space character. IDaSS places no restrictions on the line length.
Each line starts with a 'switch' character. Following this a second or even third switch character may be present, depending on what the line means. Optionally, one or more data items may be placed following the switch character(s). Each data item (including the last) is followed by at least one space character. For two of the switch characters (the single and double quote), the remainder of the line is pure text, not organised into data items. For memories, the line following the 'L' switch character contains the DOS path to the contents file.
For a complete description of the file syntax, see [Vers93]. Some examples of IDaSS design file syntax are shown below.

State machine controller

```
<stateMachineController> ::=
   '#StateControl'<blockName>' '<CR><LF>
   '/P'<blockPos><blockSize>' '<CR><LF>
   {'"'<comment>' '<CR><LF>}
   ['Z'<stateMachineControl>]
   'S'<state><nextState><forcedState><pushState>
      <flowCommand><haltCommand><halted><resetToHalt>
      <disabled><stackMax><resetReq><forced>' '<CR><LF>
   ['T'<stackSize>{<stackEntry>}' '<CR><LF>]
   {'P'<stateNumber>' '<CR><LF>
      {'''<stateText>' '<CR><LF>} }
   '.StateControl'<blockName>' '<CR><LF>
```

In this description block the <blockPos> contains information about the place of the controller in the schematic. <blockSize> pictures the size of the rectangle. In the line following the 'Z' switch a control connector is described. The 'S' switch precedes a line containing information about the current state, the next state, etc. In the 'P' line the actual state texts are stored.

Register

```
<register> ::=
   '#Register'<blockName>' '<CR><LF>
   '/P'<blockPos><blockSize>' '<CR><LF>
   {'"'<comment>' '<CR><LF>}
   ['Z'<registerControl>]
   'V'<master><slave><aReset><sReset><function>
      <defFunc><semState><semFunc><resetReq>' '<CR><LF>
   ['I'<registerInput>]
   ['O'<registerOutput>]
   '.Register'<blockName>' '<CR><LF>
```

The 'Z', 'I' and 'O' lines describe the connectors of the register. In the line with the 'V' the actual value and function are stored.

Operator

```
<operator> ::=
   '#Operator'<blockName>' '<CR><LF>
```

```
   '/P'<blockPos><blockSize>' '<CR><LF>
   {'"'<comment>' '<CR><LF>}
   ['Z'<operatorControl>]
   {'I'<operatorInput>}
   {'O'<operatorOutput>}
   {'F'<functionName><compiledFlag>' '<CR><LF>
      {'''<functionText>' '<CR><LF>} }
   'S'<function><defFunc>' '<CR><LF>
   '.Operator'<blockName>' '<CR><LF>
```

In the operator block the function descriptions are stored in the line beginning with the 'F'. In the 'S' line the actual function and defaualt function are mentioned.

RAM

```
<RAM> ::=
   '#RAM'<blockName>' '<CR><LF>
   '/P'<BlockPos><blockSize>' '<CR><LF>
   {'"'<comment>' '<CR><LF>}
   ['Z'<RAMControl>]
   'S'<nrOfWords><wordWidth>' '<CR><LF>
   ['L'<fileSpec>' '<CR><LF>]
   {'R'<readAddressConnector>
      'D'<readDataConnector>}
   {'W'<dataMaster><addrMaster><wrEnable><defWrite>' '<CR><LF>
      'A'<writeAddressConnector>
      'D'<writeDataConnector>}
   'V'<resetValue>' '<CR><LF>
   '.RAM'<blockName>' '<CR><LF>
```

Connectors in a RAM block are described using a 'R'-'D' or a 'A'-'D' combination.

Schematic

```
<schematic> ::=
   '#Superblock'<blockName>' '<CR><LF>
   '/P'<blockPos><blockSize>' '<CR><LF>
   {'"'<comment>' '<CR><LF>}
   '/W'<sheetOrigin><sheetSize>' '<CR><LF>
   {'C'<connector>}
   {<internalBlockWithViewers>}
   {<internalBusWithViewers>}
   '.Superblock'<blockName>' '<CR><LF>
```

Beneath the lines containing size and placement information the internal blocks and the buses are described. Essentially all bus descriptions come after the block descriptions.

Buses

```
<bus> ::=
   '#Bus'<busName>' '<CR><LF>
   {'"'<comment>' '<CR><LF>}
   {'C'<refNumber><blockName><connectorName>' '<CR><LF>}
   {'/N'<refNumber><nodePosition>' '<CR><LF>}
   {'/S'<beginNode><endNode>' '<CR><LF>
      {<nameViewer>|<valueViewer>} }
   '.Bus'<busName>' '<CR><LF>
```

Within a bus description reference numbers are assigned to connectors of several blocks and to node positions in the schematic. The bus definition is completed by describing the bus segments between the reference numbers.

4. SimDes to IDaSS layout generation

4.1 General aspects of the translation

After describing the source and target environment, the next step is the actual translation between the tools of level 1 and 2 in the PASS hierarchy. The desired goal is to create a tool that automatically generates an IDaSS design from the processor description written in SimDes. This means a translation from one ASCII textfile to another one, so that the last one complies to the IDaSS design file format. In table 5 a first possible relation between each SimDes description block and its IDaSS equivalent is shown. The 'Statistics' blocks will be ignored during translation.

Table 5: SimDes - IDaSS relation

SimGen description block	IDaSS description block
RAM	Memory: RAM
Pc	Register, reserved for the PC, with accompanying buffers
FetchBuffer	Register(s) used as FB
Registers (global and local)	Registers
Ports	Superconnectors, externally connected to I/O registers
Functions and Instructions	Combination of Operators and Controllers

The most important difference between the processor descriptions for SimGen and for IDaSS is the presence of graphical information in the IDaSS description. Since there is absolutely no graphical information included in the SimDes file, this must be added by the tool that generates the IDaSS description. The result of the translation must be a textfile that, after loading it in IDaSS, shows a layout design of the processor. The instruction set is implemented and the designer can start simulating using the 'step' option.
In the rest of this report the tool that takes care of the translation will be referred to using the name:

"SimIdass"

The contents of this chapter discuss the generation of a graphical layout using the resources description in SimDes. But first another aspect of the translation must be treated: the SimDes to IDaSS compiler front-end.

4.2 SimIdass front-end

After a processor designer has written his SimDes description file, he might for some reason try to skip the SimGen simulation generation and start using SimIdass immediately. With skipping

SimGen the checking for grammatical and syntactical errors in the description will be skipped too. There are two solutions to this problem, both graphically represented in figure 11.

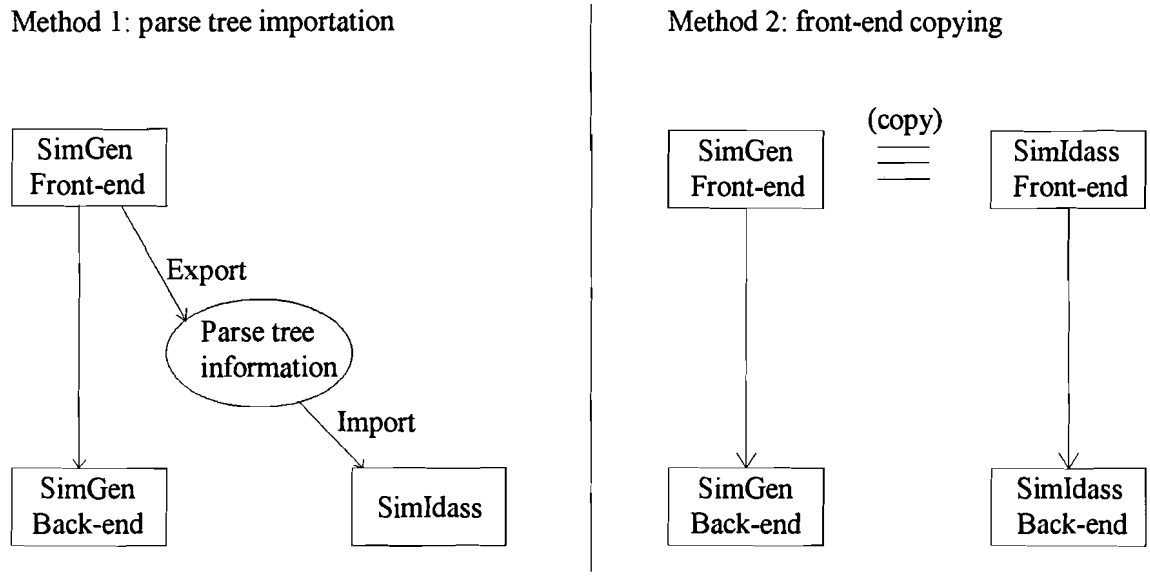

Figure 11: Grammatical and syntactical error checking

The first option is to provide SimIdass with its own front-end for error checking. The other possibility is simply not to tolerate skipping SimGen. In this case the parse tree, constructed from the processor description, could be used instead of the original description file as input for SimIdass, with the advantage that all information has been organized and can be looked up easily. To make this possible the SimGen tool would have to be changed so that the parse tree information can be stored in a file together with additional information like a resource name lookup table. The main advantage of this method is that the actual SimDes to IDaSS translation takes less time, because there is no need for scanning and parsing. On the other hand this option implies an adjustment to SimGen, which may not be easy to program. Additionally there is the disadvantage of being obliged to execute the back-end of SimGen, even if this is not desirable. That is why the other option, providing SimIdass with its own front-end, seems to be preferable.
The programming of a front-end for SimIdass is not very difficult, because it can be done by copying some of the SimGen modules. By examining the working of each module a separation can be made between the standard SimGen sources which are needed for this front-end and those which are not. It is obvious that the two input language dependent files, "SimGen.rex" and "SimGen.lalr", should be accessible for SimIdass. Furthermore all files must be copied from the modules which perform the following tasks: scanning, parsing, tree building and tree checking. The files that involve with this modules are:

SimGen.rex	CheckType.h
Token.h	CheckFunc.h
ScanType.h	CheckFunc.c
Scanfunc.h	Check1Func.c
Scanfunc.c	Check2Func.c
SimGen.lalr	Check3Func.c
Treetype.h	
Buildfunc.h	
Buildfunc.c	

These files are copied to a special directory for SimIdass.
The result of executing the front-end is a datastructure, filled with the complete parse tree, and tables containing all additional information needed for further translation. The structure of the parse tree is appointed in the file "TreeType.h", as described in appendix C. The directory must be completed by additional files performing the actual translation functions.

4.3 Projection of resources on layout blocks

After executing the front-end, SimIdass will generate a graphical layout according to the processor description. The first step in the SimIdass back-end generation is to research the possibility of automatic layout generation. In section 4.1 a table was presented which shows what the projection of SimDes on IDaSS could look like. In this section it will be made clear that it is possible indeed to realize this projection.

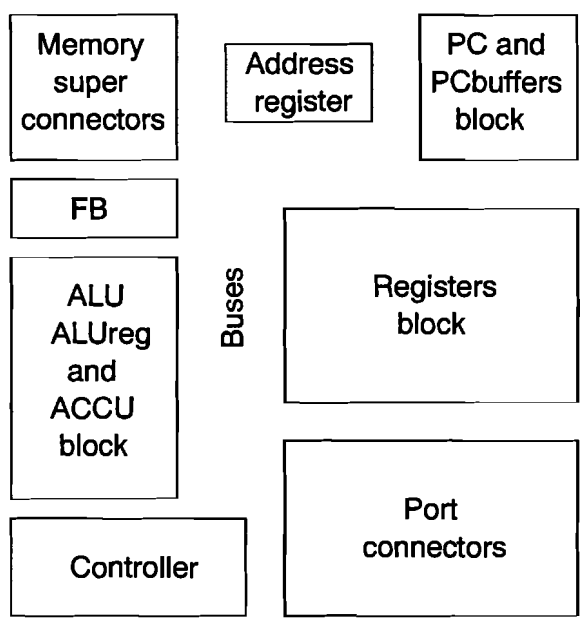

Figure 12: Block model for processor layout

4.3.1 Preamble to layout projection

Before evincing what the actual translation will look like, it is necessary to make some remarks about some aspects of the layout. The first one is the memory translation problem. In IDaSS a memory block can be placed in a schematic. This memory is meant to be 'on-chip-memory' and may contain 2048 words of 64 bits each. The memory block in SimDes however defines a memory that is not part of the processor as such. The size of this resource is not limited, but in most cases the block will contain much more than 2048 words. It should be obvious that the denotation of the two memories are completely different. Therefore the SimDes memory resource will not be translated to a block in the processor schematic, but to a number of superconnectors, that is externally connected to a memory schematic.

The second note is about the IDaSS buswidth. In IDaSS connectors of several blocks can be connected by buses on condition that the bitwidth of all connectors connected to one bus is the same. In SimDes there are no restrictions for data width at all. During translation this must be taken into account.

4.3.2 Standard processor layout type

For the actual design layout in IDaSS a standard processor structure is chosen (figure 12). This model exists of a number of resources and one ALU operator that can get data from the resources via a data-in bus and can store data in the resources via a data-out bus. Among the resources are the registers, the PC register, the fetchbuffer, the superconnectors to the I/O ports and those to the memory block. All resources have a bitwidth according to the declarations in the SimDes description. To connect all blocks with the two data buses it is necessary to provide each register and each superconnector with a buffer or an operator that takes care of adapting the bitwidth of the data. The width of the two buses in the schematic will be set to the maximum of all bitwidth declarations for the resources.
With regard to the new information in this section it is possible to alter the contents of table 5 into that of table 6.

Table 6: More detailed SimDes - IDaSS relation

SimGen description block	IDaSS description block
RAM	Superconnectors (address, data read, data write, control) connected to an external memory
Pc	Register, reserved for the PC, with outputbuffers and eventually an input bitwidth operator
FetchBuffer	Register(s) used as FB, eventually with bitwidth adaptor
Registers (global and local)	Registers, with bitwidth adaptors where necessary
Ports	Superconnectors connected to external registers (required for simulation). Buffering is necessary
Functions and Instructions	Combination of one operator and one controller

There are four possibilities of which resource determines the buswidth:

- one of the registers, because of its bitwidth
- the programcounter, because of its bitwidth
- the RAM, because of its bitwidth
- the RAM, because of its addressbuswidth

The actual layouttype influences the placement of the blocks in the IDaSS schematic, especially

the datawidth operators. SimIdass places a value in a variable called "lotype" (short for layout type). Subsequently it counts the number of (global and local) registers with a bitwidth that differs from the maximum bitwidth (counter: c0) and also the number of registers with a bitwidth that equals the maximum (counter: c1). The meaning of the "lotype" value is shown in table 7.
The program makes its placement decisions on the basis of the "lotype" and "c0" values. Some of the building blocks are placed by SimIdass at standard coordinates. The complete placement process is shown below:

- place memory connectors, starting with the "adr" connector at (2,2)
- place FB sub-schematic at (2,34)
- place AREG register at (28,10)
- place PC at (112,2); PC-buffers at (80,2) and (80,10)
- test the values of "lotype" and "c0"
- place other blocks:
  - if `lotype` even then place DtoA operator between databus and AREG (52,10)
  - if `lotype` > 3 then place DtoM operator between databus and memory write connector (28,18), and MtoD operator between memory read port (and FB) and databus (28,26)
  - if `lotype` < 2 or `lotype` > 5 then place DtoPC operator between databus and PC (136,2)
  - if `c0` = 0 then place registers at (64,y) with y from 26 to (18+8*(c0+c1))
    in the other case, `c0` != 0 (at least one of the registers has a width smaller than the buswidth) then:
    place registers more to the right, starting at (96,y)
    place operators where needed, at (64,y) and (128,y)

Table 7: "lotype" values

"lotype" value	maximum width equals:	datawidth operators needed with:
0	memory width	PC, address connector, registers (if c0 > 0)
1	memory width & address width	PC, registers (if c0 > 0)
2	memory width & PC width	address connector, registers (if c0 > 0)
3	memory width & PC width & address width	registers (if c0 > 0)
4	PC width	data connectors, address connectors, registers (if c0 > 0)
5	PC width & address width	data connectors, registers (if c0 > 0)
6	some register width	PC, data connectors, address connectors, registers (if c0 > 0)
7	address width	PC, data connectors, registers (if c0 > 0)

Some examples of the aim of the actual text translation of resources will be presented in the next sub-sections.

4.3.3 Memory translation

As described before, the SimDes Ram block does not belong to the actual processor but will be directly accessible to it. In the SimDes description it is added to the processor resources, and looks like:

```
|[ memory   :  M  [4194304]  {16} ]|    # 4 Mb RAM memory
```

In the IDaSS file it will be subtracted from the processor and form its own schematic. This schematic contains superconnectors to connect it to the processor block, and the RAM memory itself (figure 13). A problem is the limit to the RAM depth in IDaSS: 2048 words. This is the reason why the SimIdass has the same limit during translation. All memory depths exceeding this limit will be diminished. In this example translation will result in:

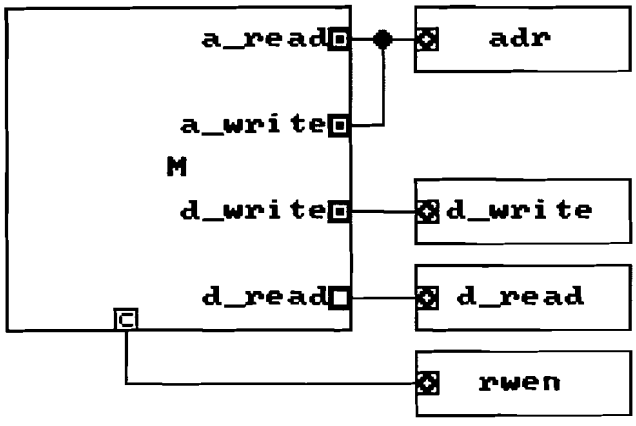

Figure 13: Standard memory block

```
#RAM M                          " block header
/P2@2 32@30                     " position in the memory schematic
ZC*c 0 -3W2 1                   " control connector
'(1,0)                          " contr. conn. function description
'10b write.
'01b enable
/P10@28
S2048 16                        " memory size and width
RIa_read 1 -3W10                " read address input
/P30@2 -12@0
DTd_read 1 -3W16 0 0              " read data output
/P30@26 -12@0
W-3 -3 0 0
AIa_write 1 -3W10             " write address input
/P30@10 -14@0
DId_write 1 -3W16             " write data input
/P30@18 -14@0
V0
.RAM M
```

By adding additional lines for the position of the memory schematic in the toplevel and for the superconnectors and internal buses the memory description will be completed.

4.3.4 PC translation

The description of the programcounter in SimDes look like:

```
|[ PC    :  PC  { 16 }  =  { 255 } ]|
```

The width of this counter is 16 bits and its initial value is 255. This line will be translated to an IDaSS register block that will have the following form:

```
#Register PC
/P112@2 18@6
V255W16 255W16 255W16 255W16 def hold 0 holdSem 0
II*i 0 -3W16
/P16@2 ?
OO*o 0 255W16
```

```
/P0@2 ?
.Register PC
```

Two buffers automatically will be added, one for the address bus and one for the data bus. If the databus width is larger than the PC bitwidth an operator will also be placed before the PC input connector.

4.3.5 Register translation

Just like the programcounter each register description line will be transformed to a layout block in IDaSS. An example of a description line:

```
|[ registers    :
   ...
   AF  { 32 }  =  { 0 } ;
   ...
]|
```

The result of translating this 32-bits AF register with initial value 0 will be:

```
#Register AF
/P128@96 26@6
V0W32 0W32 0W32 0W32 def hold 0 holdSem 0
II*i 0 -3W32
/P24@2 ?
OT*o 0 0W32 0 0
/P0@2 ?
.Register AF
```

If the width of internal bus exceeds that of this register, additional operators called AFtoB and BtoAF will be placed before and after the register connectors.
It is also possible in SimDes to describe an array of registers. In that case this will be transformed to a register schematic within the processor schematic. In this sub-schematic the registers that are part of the array will have their places.

4.3.6 Fetchbuffer translation

In SimDes the fetchbuffer always is described like an array of words, even if it exists of only one word like in this example:

```
|[ fetchbuffer :  FB  [ 1 ]  { 16 } ]|
```

The width of the fetchbuffer words must always be exactly the same as that of the memory.
In IDaSS the fetchbuffer will exist of a sub-schematic in which the FB registers are located. This is described in the next piece out of the IDaSS file.

```
#SuperBlock FB              " superblock header
/P2@34 20@10                " position in the processor schematic
/W0@0 65@26
CBin 1 ?                    " superconnector "in"
/P18@2 -4@0
CBout 1 ?                   " superconnector "out"
/P18@6 -6@0
#SuperConnector in
```

```
/P2@2 10@6
CBin 1 ?
/P8@2 ?
.SuperConnector in
#SuperConnector out
/P50@2 10@6
CBout 1 ?
/P8@2 ?
.SuperConnector out
#Register FB_0            " actual FB register
/P18@2 26@6
V0W16 0W16 0W16 0W16 def hold 1 holdSem 0
II*i 0 -3W16
/P0@2 ?
OT*o 0 -3W16 0 0
/P24@2 ?
.Register FB_0
#Bus fbin                 " FB-internal input bus
C1 in in
C2 FB_0 *i
/S1 2
.Bus fbin
#Bus fbout                " FB-internal output bus
C1 out out
C2 FB_0 *o
/S2 1
.Bus fbout
.SuperBlock FB
```

The text describes a fetchbuffer with only one internal register. In the next figure the contents of the fetchbuffer sub-schematic are shown, in which the FB exists of three registers.

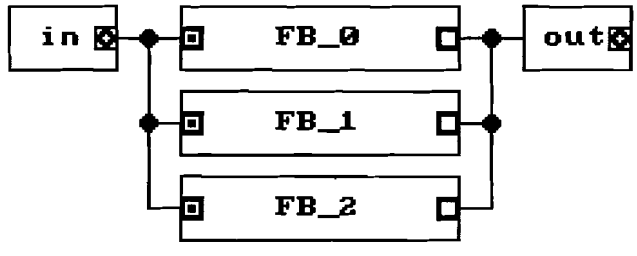

Figure 14: Fetchbuffer schematic

4.3.7 Instruction set implementation blocks

If a simulation is started on a design existing of just resource blocks, nothing would happen. There is need for some standard building blocks which take care of controlling the dataflow through the schematic. Although there is no solution yet for the problem how to implement the instruction set functions, an assumption can be made that this implementation will need at least two blocks: an ALU operator and a state machine controller. For complete ALU-like operation the operator needs to be supplied with additional registers: an ACCU as input buffer, an ALUReg as output buffer and a TEMP register for longer term storage. The ALU related blocks all have a reserved place in the schematic.

4.3.8 Adding buses

When all resources are placed, it is time to connect the blocks using buses. Not only the two databuses need to be defined, also buses are required to connect resources with their accompanying bitwidth adaptors. It is not very difficult to find out which connectors must be connected to which bus. The hard part of defining a bus is to appoint the coordinates of each bus node in the schematic. This coordinates depend on the placement of the resource blocks. A special part of the SimIdass program will take care of the bus allocation problem. Buses between resources and their accompanying operators are easy to define. It is more complex to calculate the exact coordinates of all nodes of the databuses . This is done in a similar way as the placement of the building blocks. Most of the coordinates are fixed, the others depend on the values of "lotype", "c0" and "c1". So the procedure for the graphical definition of the ALUout databuses will look like:

- declaration of the connectors that are to be connected by this bus:
  - standard connectors: "ALU o", "ALUREG *i", "TEMP *i", "<port>b w" for each port
  - connectors, dependent on "lotype":
    if `lotype` > 3 then "DtoM i" else "d_write d_write"
    if `lotype` even then"DtoA i" else "AREG *i"
    if `lotype` < 2 or `lotype` > 5 then "DtoPC i" else "PC *i"
    for each register: "Dto<name>" if width < buswidth
    "<name> *i" if width = buswidth
- calculation of node-coordinates:
  - standard nodes
  - coordinates, dependent on "lotype"; the number of registers and ports determines the actual number of nodes needed
- definition of the segments between nodes and connectors

The actual SimIdass code for this bus definition is included in appendix D. For the ALUin databus a similar procedure is used.

Now that all major layout parts have been discussed two examples of completed layout generation are presented. The first example is derived from a description where all buses have a width of 16 bits and so have all resources (see figure 15). The other example shows a design where the buswidth is equal to the bitwidth of the largest registers. All resources with smaller bitwidth are connected to the databuses via bitwidth adaption operators (figure 16). Notice the difference between the two figures, especially the bus node coordinates.

4.4 SimIdass translation program

The translation program is written in the language C, and is built up from several functions each of which performs a specific task. Before writing a group of lines in the design storage file, some values will have to be retrieved from the parse tree. This can simply be done by looking up the required values in the datastructure in which the parse tree was stored. When all values for one resource are retrieved, the IDaSS description lines can be sent to the file.

4.4.1 Writing the design file

To make a file accessible for writing, it has to be opened first. After that SimIdass uses the file to

send lines to. The writing of the lines is done by using the C command "fprintf". In the SimIdass program code this command is used very often, but that makes the text easy to understand. An example of a piece of code text is:

```
fprintf(idassfile,"#Operator DtoPC \n/P136@2 18@6 \n");
fprintf(idassfile,"IIi 1 -3W%d \n/P16@2 -2@0 \n",buswidth);
fprintf(idassfile,"OOo 1 -3W%d \n/P0@2 2@0 \n",width);
fprintf(idassfile,"Fadjust 1 \n'o := i width: %d \n",width);
fprintf(idassfile,"S0 adjust \n.Operator DtoPC \n");
```

This results in a description of a buswidth to PC bitwidth adjustment operator.

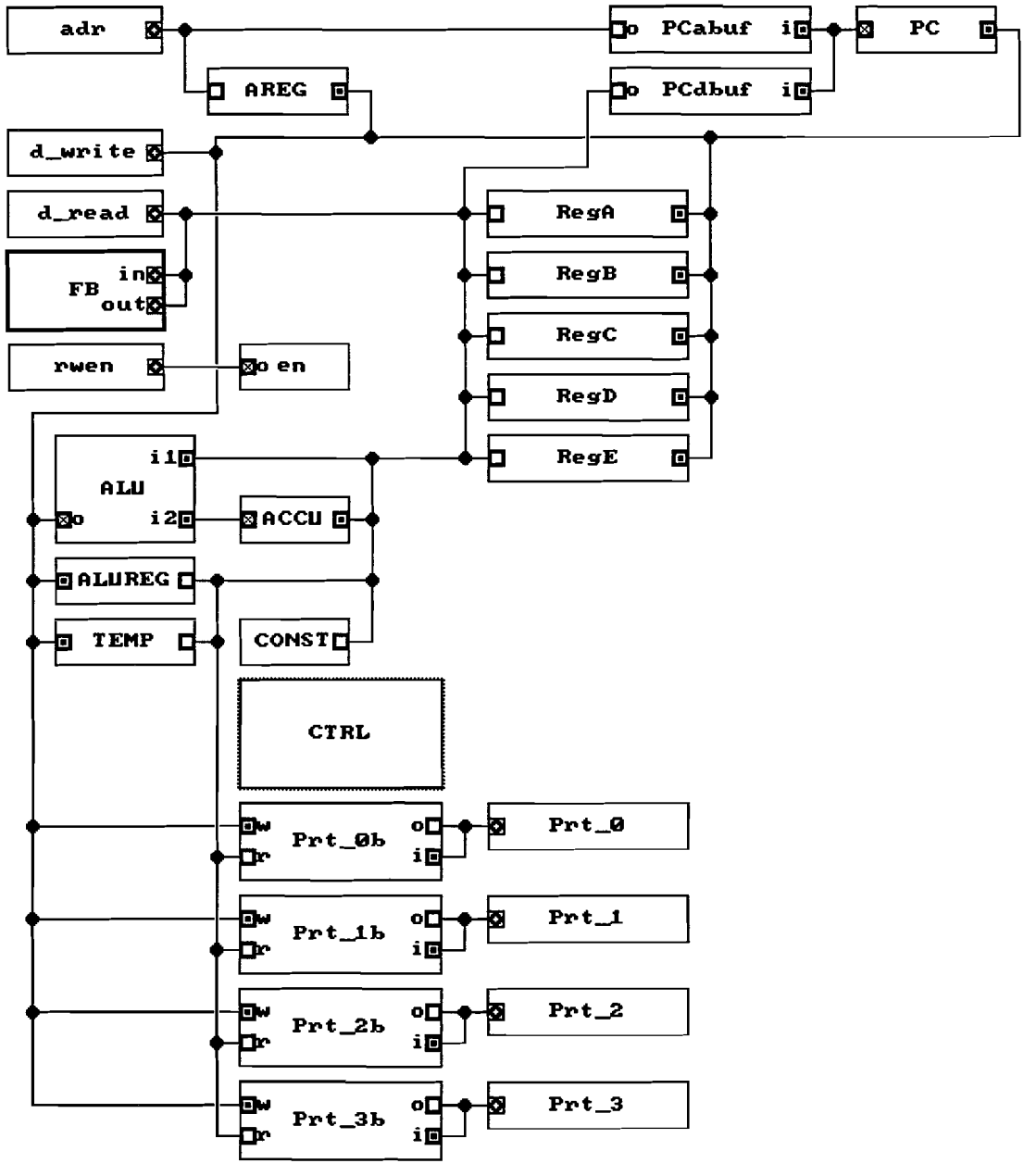

Figure 15: Layout example 1

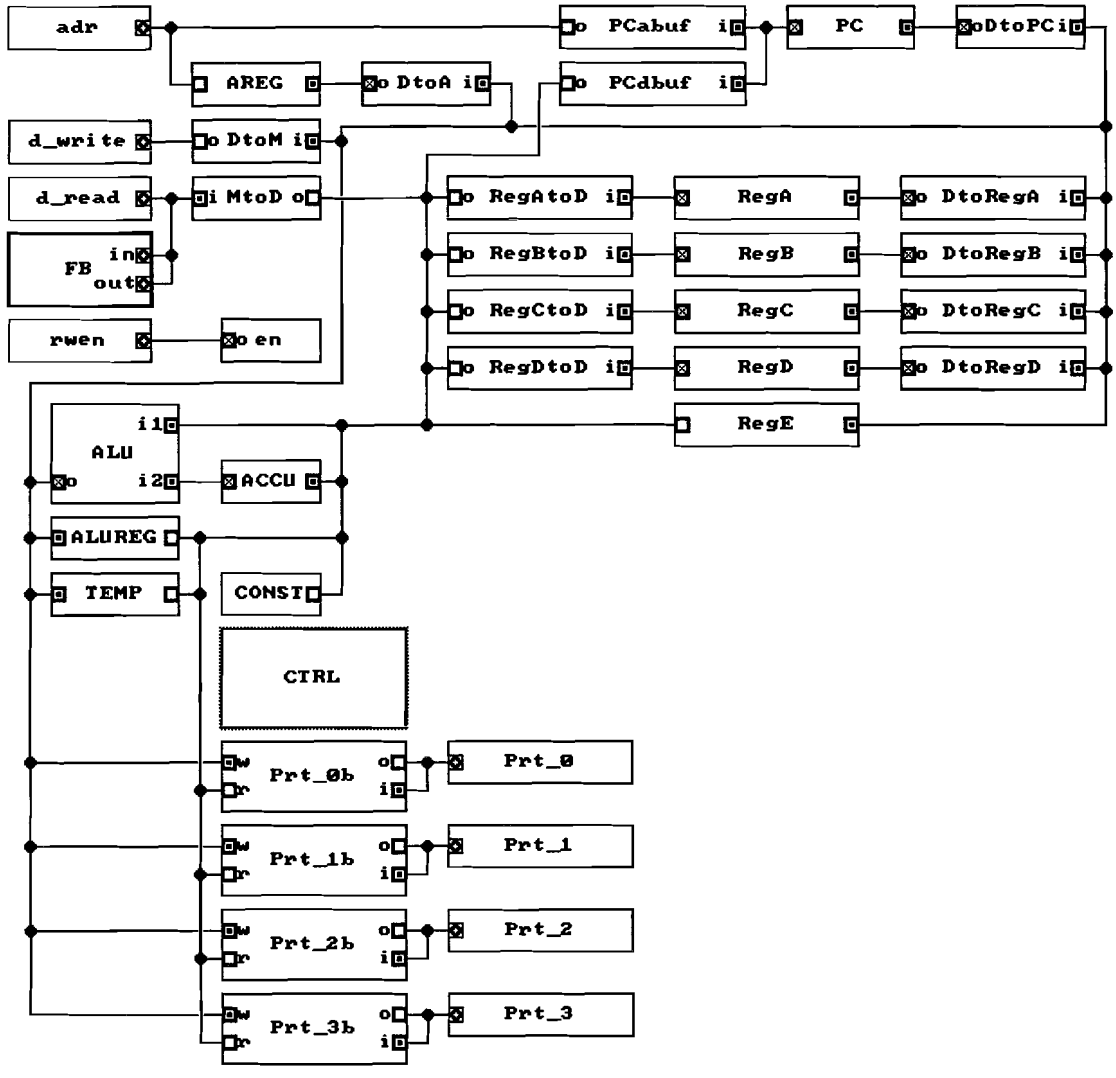

Figure 16: Layout example 2

4.4.2 SimIdass back-end structure

The SimIdass back-end code has almost the same hierarchical function structure as that in the front-end parse tree. The reason of this is that it is easier to read through the program text, especially when one is familiar with the tree structure already. The layout generation functions are all included in one and the same source file: "WriteIdass.c" . In figure 17 the structure of the code functions is presented. For a brief description of each function see appendix E.

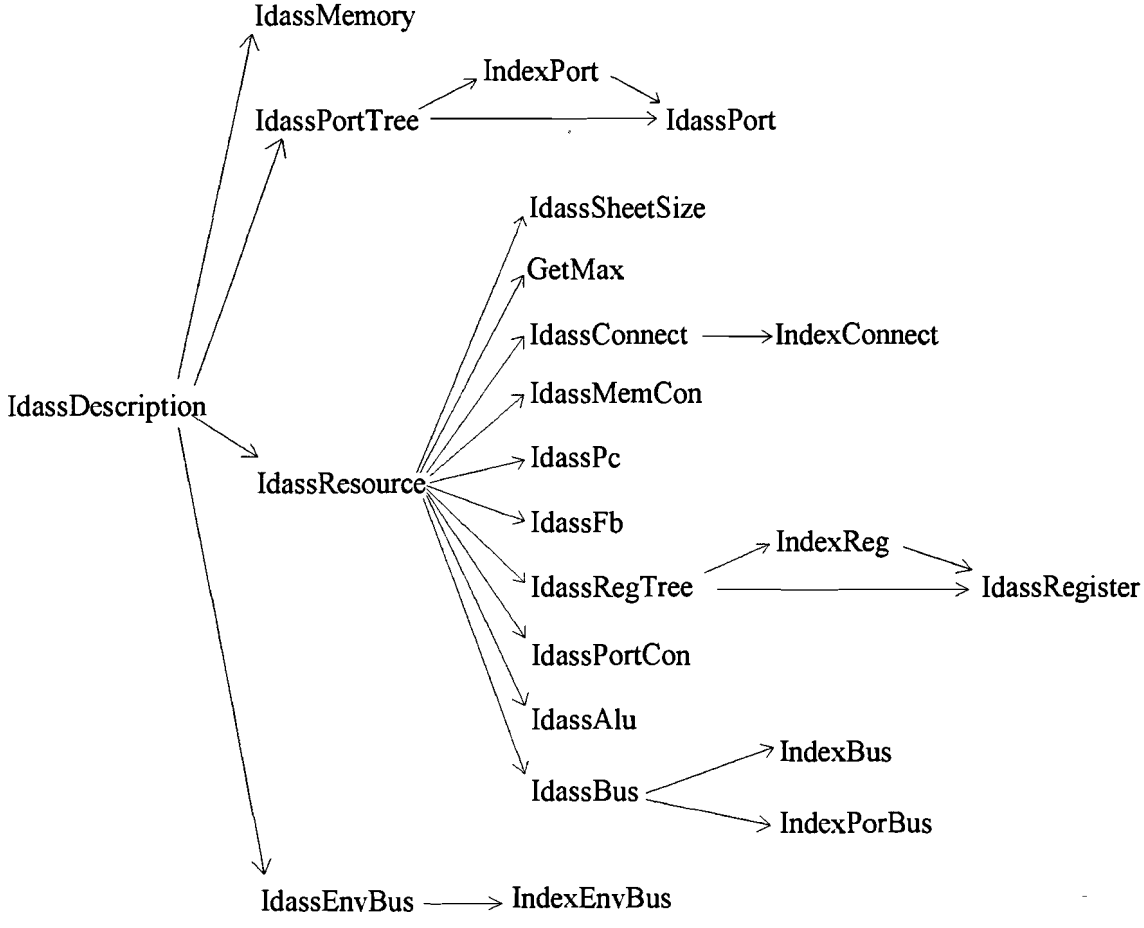

Figure 17: SimIdass back-end function structure

One extra file is constructed especially for connecting the SimIdass front-end and back-end to each other. This file is called "SimIdDriver.c" . If the SimIdass file itself is not generated yet, it can be done in a directory where all module files are put together. This refers to the files mentioned in section 4.2 and to "WriteIdass.c" and "SimIdDriver.c" . There are two more files that have to be present in the directory: "BlockFunc.h" and "BlockFunc.c" . These files are part of a first attempt to implement the instruction set functions, and are used by SimIdass to define the ALU and controller blocks. More on these files follows in chapter 5.

4.4.3 How to use SimIdass

Not only the code structure, but also the way SimIdass is used resembles that of SimGen. All files

related to SimIdass must be present in one directory. If this is complied with, SimIdass has to be generated just once. For the SimIdass generation a special Makefile was constructed, that must be present in the same directory. The generation process will be started after typing:

```
make SimIdass
```

Like the SimGen generation, the makefile will create a scanner and a parser first using the SimGen.rex and SimGen.lalr files. After that it will compile all related code files and build SimIdass out of them, resulting in one single executable file called "SimIdass".
Presuming a processor description file called "Test", SimIdass is able to translate it to one or more IDaSS design files. The number of files depends on the number of implementation blocks within the description. The translation can be started using the command:

```
SimIdass < Test
```

If logs are activated, information about the front-end execution will be placed in the debug-files. If a description is incorrect, syntactically or semantically, the code-generation process will be aborted. If a description passes the syntactic and semantic verification, the IDaSS design files will be generated for each processor implementation in the SimDes description. The files will have names like:

<Implementation_name>.des

The next step is to read the contents of the generated file in IDaSS. After starting up the IDaSS tool in a Smalltalk environment, one of the menu options is to load a complete schematic from file. The directory where the design file is to be found can be entered here. By selecting the right design file IDaSS will read its contents and finally show the toplevel of the schematic hierarchy. This toplevel exists of three parts: one is the memory block which is taken apart from the processor; the second is the processor block itself; a group of registers connected to the processor block by buses represent the I/O ports and form the third part (see the example in figure 18).

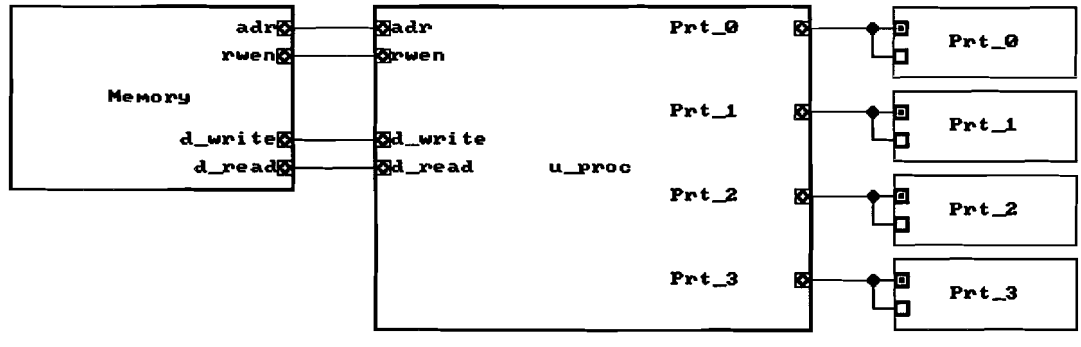

Figure 18: IDaSS toplevel example

5. Instruction set implementation

In the previous chapter is has turned out to be possible to transform any SimDes processor description to an IDaSS layout design. The next step shall be the translation of the instruction set into IDaSS operator functions and state machine descriptions. This chapter is a disquisition of a research on this subject; it pictures the problems that may appear during creation of an instruction set implementation tool, and describes possible solutions to this problems.

5.1 Instruction set implementation and SimIdass

This chapter goes into problem solving during the implementation of the instruction set. The implementation method is dependent on the model used for layout generation. This will be called 'layout model dependent'. In case of SimIdass the design layout model is a 'standard layout model' because of the standard division of the layout parts.

5.1.1 Method of implementation related to SimIdass

The most obvious method of implementation that uses a standard layout model, like that of SimIdass, is translating the instruction set functions into descriptions for one ALU operator and one controller. In the ALU operator all functions must be programmed that are needed for executing each instruction set statement. The controller must contain information about the assignment of the proper functions to the ALU operator and to all other resources in the schematic. In this controller the traditional 'Von Neumann' stages will be represented. During the 'Fetch' stage the opcode will have to be loaded from memory and stored in the fetchbuffer. The states of the controller that take care of this are dependent on the layout structure only, not on the instruction set. The 'Decode' and 'Execute' stages are carried out simultaneously.
A simple example is the translation of an 'ADI' (Add Immediate) instruction. Suppose an instruction description that look like:

```
|[ Instruction   :
   ...
   ADI   ( 198 | )
   {
      A  =  A  +  M [ PC ]
   }
   ...
]|
```

Each state of the 'Execute' stage starts with a test of the opcode. In this example the fetchbuffer exists of one single register. The instruction 'ADI' only will be executed if the opcode has the value 198. To enable the execution of this instruction the ALU operator must have an 'add' function at its disposal. In the design storage file this would look like:

```
#Operator ALU
...
Fadd 1
'o := i1 + i2
...
.Operator ALU
```

During the 'Execute' stage the controller will have to assign the right functions to the resources.

In this example the following text must be present in the storage file:

```
#StateControl
...
P..                          #  state number ..
'Exec0:
'[ FB\FB_0                   #  test FB opcode
'| 11000110b A enable;          #  if FB_0 = 198 then execute
'            ACCU load;          'ADI' instruction
'            -> Exec1            #  goto Exec1 for next step of execution
'| ...                       #  else if ...
']
P..                          #  next state number
'Exec1:
'[FB\FB_0                    #  test FB opcode
'| 11000110b PCabuf enable;    #  if FB_0 = 198 the execute
'            en ren;             rest of instruction
'            MtoD enable;
'            ALU add;
'            A load;
'            PC inc;
'            -> Fetch0           # end of execute stage
'| ...                       # else if ...
']
...
.StateControl CTRL
```

In the 'Exec0' state the value of register A will be stored in the ACCU register. During the next state the value of the memory word with address [PC] will be added to that of the ACCU, and the result will be put back in A.
After programming all functions in the operator and designing a controller that will coordinate the function assignment, the complete instruction set is represented by these two blocks.

5.1.2 Difficulties with SimIdass implementation method

The implementation method as described in the previous sub-section will probably be possible to realize but there are some aspects which must be mentioned because they may cause some problems. It will be difficult to derive all functions for the ALU and all state descriptions for the controller from the processor description, certainly when this must be done automatically by a tool. The main problem with constructing such a tool is the complexity of the translation. But there is more to it:

* A possible question is if there is some kind of limit to the number of functions one operator in IDaSS may have at its disposal. Within IDaSS there is no such limit, but when an instruction set is very extensive the number of functions can be enormous and that might cause troubles during loading in IDaSS of the design file information.

* Another aspect is the question to what extent the IDaSS design will be optimal after using the SimIdass layout model for translation. The fact that all functions will have to be performed by one and the same operator influences the timed needed for instruction execution. If several function must be executed for one instruction, this takes a while because all functions are carried out sequentially. In principle this is just like a sequential processor is meant to be, but in this case it goes further. The most expressive example is that of setting and resetting flags.
  In almost every processor description a special kind of resource contains the flag bits; this is the status word. This register exists of a number of flag bits which are influenced by each

instruction. In SimDes this resource is treated as a normal register. If this is also done in IDaSS this is very inefficient. A number of functions must be executed specially for setting or resetting the flags right after executing the actual instruction statements. In a more usual processor implementation the flag register is revised at once and in parallel with the other instruction statements.

The above-mentioned problems are mostly specific for the chosen design layout model, on which the instruction set implementation is based, in this case the 'standard layout model' (figure 12). Therefore alternatives for the model in SimIdass have been searched for. In what extent this alternatives deal with this problems will be explained in the next section.

5.2 Layout model alternatives

There are possibilities to change the basic model for the IDaSS layout so that some of the problems can be solved. The first option is to create an automatic data path synthesis tool. The other method sticks to the idea of a standard layout but differs in one way with the SimIdass model: it has not one, but many ALU-like operators.

5.2.1 Automatic data path synthesis

The first layout alternative is the most complex one. It consists of a tool that will automatically derive the ideal data path from the instruction set description. This method does not hold on to any standard model; the resource layout is totally dependent on the data path derived by the tool.
A lot of literature is written about this subject. General aspects of this method are described in for example [Diet75] and [Beik90]. The first article starts with a definition of the levels between our original PASS levels 1 and 2. It mentions an 'organization level' at which the data path is derived from the description. This level is followed by the 'algorithm level' where algorithms are constructed to combine information from the resource definitions and the constructed data path. With the help of this processor dependent algorithms the design at the next level, the 'register transfer level' can be produced. In the other article steps are mentioned that are part of the complete data path synthesis process. These steps are specified as:

- generation of an internal representation for the behavioral description
- scheduling of the operations in the instruction set
- allocation of the hardware building blocks
- synthesis of a unit that controls the hardware functions

The allocation of the hardware building blocks can be derived partially from the definition in SimDes. The other parts, the operators and the interconnections, have to be added automatically.
It is obvious that this method is far more complex than the one that is combined with the SimIdass layout model. All layout specific problems mentioned before can be solved using this synthesis process, but it is not for certain that it is possible to create such a tool in an acceptable period of time. Therefore another implementation method was thought out: an implementation based on a distributed ALU operator model.

5.2.2 Distributed ALU operator model

A way to overcome the problem of creating an ALU with a profusion of function definitions, is to make use of more than one operator. It might be possible to combine this operators with the bitwidth adjustment operators and buffers which must be added to the resources. Instead of declaring one function just to connect a resource to the internal bus, all ALU functions which concern assignment to this resource could be added to the description of this operator.
A very powerful design can be obtained if the internal bus structure is also changed. Since the model is not restricted to only one ALU operator, it is recommendable to let the number of internal buses depend on the data flow structure in the instruction set definition. This can be done in several ways. The first method leans towards the datapath synthesis model; information about the number of buses and the connections between the buses and connectors will be retrieved from the statement descriptions.
A more simple method is to let only the number of buses depend on the statements. Per statement the program can check which of the buses is free at that time, and makes a connection. The number of different buses each operator will be connected with appoints the number of connectors needed for this operator.
With both methods it is necessary to define at least the global placement for the IDaSS blocks. Later on the exact coordinates can be calculated from the number of connectors per operator. To avoid serious routing problems for the databuses, it is advisable to place the resources in a row, so that the buses can be placed side by side in parallel with the blocks. In figure 19 an example of a schematic according to this model is shown.
An additional advantage of this model is that the flag register has its own operator with the special flag setting functions. Now the flag setting and the other instruction statements can be executed simultaneously in spite of the fact that the translation program does not need to know the difference between an ordinary register and the status word.
Reverting to the example of the previous section, the 'ADI' instruction, the implementation of this function would come down to defining an 'add' function for the operator that is placed in front of the A register. The controller will assign this function to the operator at the right time, so that it can calculate the new value for A from its input connector values. One of the connectors will have to be connected with the output connector of the memory.
The distributed operator layout alternative appears to be the ideal solution to the problems mentioned in 5.1.2 . To create a tool however which automatically derives the IDaSS design according to this model from the SimDes description is far more complicated than in case of the SimIdass model. Not only the operator functions and controller states must be derived from the description, also the functions must be placed in the right operators. Additionally the bus structure must be computed in a different way, because the number of buses that are used depends on the instruction set contents. It is impossible to create a model for this layout method in which the exact placement of the resources is known. All placements depend on the bus structure and on the number of input and output connectors each operator needs.

5.3 General instruction set implementation concept

There are three different models described so far, that can be used for the SimDes to IDaSS translation. None of the instruction set implementation methods has been programmed yet. To make at least one step in that direction a general method has been thought out to bring some structure in the implementation step. This method is independent of the layout model that is used for the resource translation and consists of three stages, graphically visualized by figure 20.

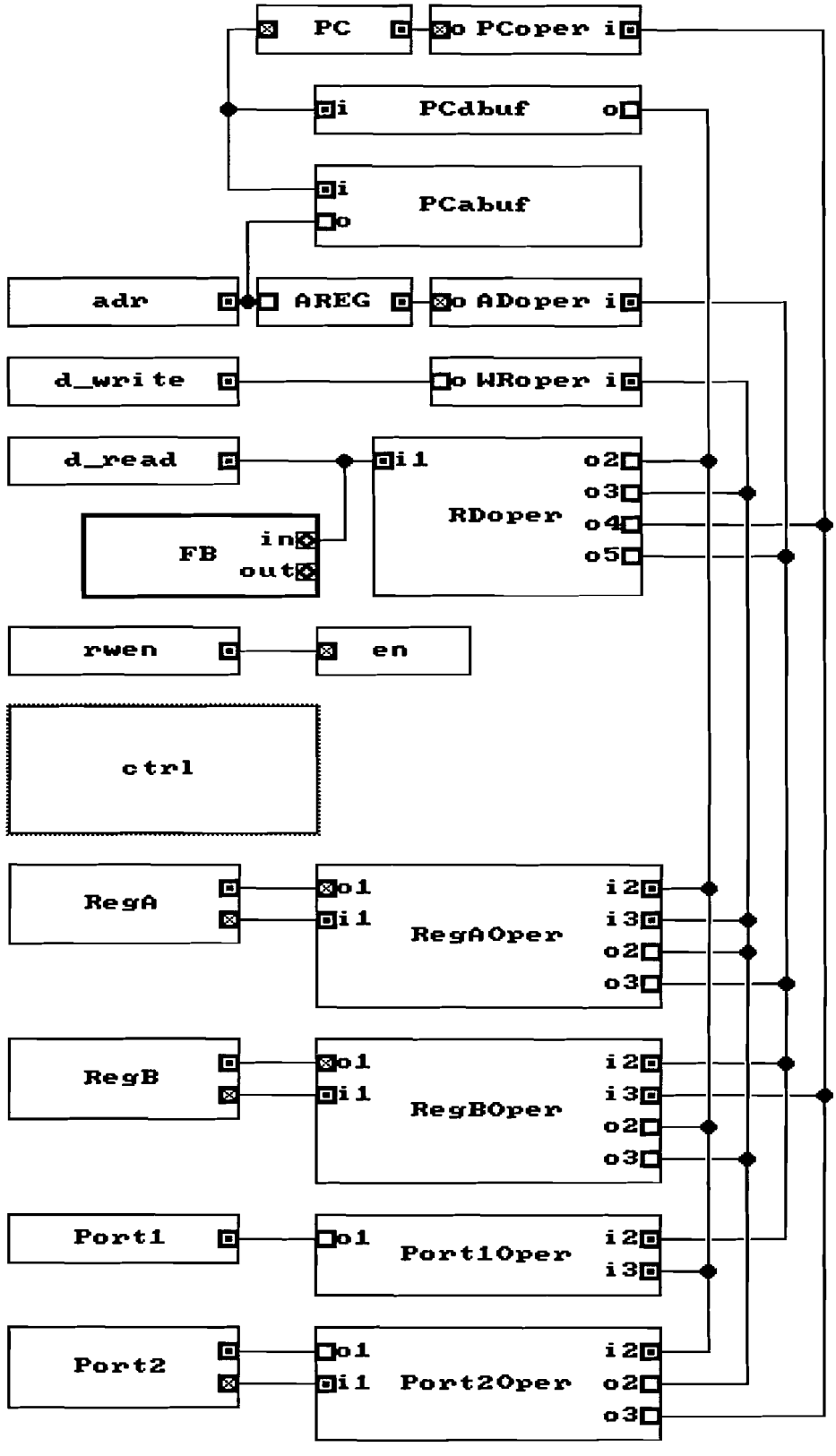

Figure 19: Distributed operator layout

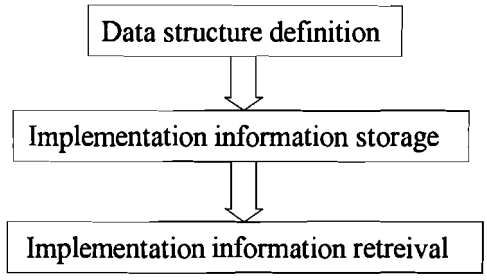

Figure 20: General implementation proces stages

Data structure definition

The first stage must be gone through before programming the implementation part in the translation tool. A data structure is defined in a way so that all information, needed for writing the instruction set implementation specific parts in the IDaSS file, can be retrieved from it at the right time. A header file is used to describe the structure of this data type. In this structure a part is used to store operator functions information, another part is for the state controller description lines.
Figure 21 shows an example of a data type definition that might be used for this purpose; it describes a structure that is suitable for layout models like that of SimIdass, where only one operator and one controller is needed, but also models with more than one operator or controller are possible. The formal definition of the data structure is presented in appendix F. This structure is contained in a file called "BlockFunc.h" .
This data structure is only suitable for models where all graphical information can be calculated from the resource definitions, not from the instruction set information. In all other cases the placement of the blocks and buses depends on the instruction set definition. Therefore special fields will have to be included in the data structure in which the graphical information can be stored. In that respect the structure will differ from that of figure 21 and appendix F. The "operator" part in the structure could for instance contain a field in which the block size is defined, and a pointer to the connector definitions for the operator. This connector definitions exist of a name and the bus to connect it with.

Implementation information storage

Using the defined data structure all function and state description lines can be stored. The SimDes to IDaSS translation tool will go through the SimDes parse tree and derive all the needed function descriptions and controller states from the instruction set definition. Line after line these descriptions will be placed in the structure, so that all lines belonging to one controller state as well as all functions belonging to one operator are grouped together. Following this method the description of the operators and controllers can be retrieved from the data structure quite easily.
Additionally, information about the graphical placement of the resources and the operators, controllers and buses might be necessary for layout generation. In case of the data path synthesis method this means that in some way information must be stored, from which the graphical coordinates for each resource in the schematic can be derived. With the other method, the operator distribution model, the required graphical information could be derived from the data

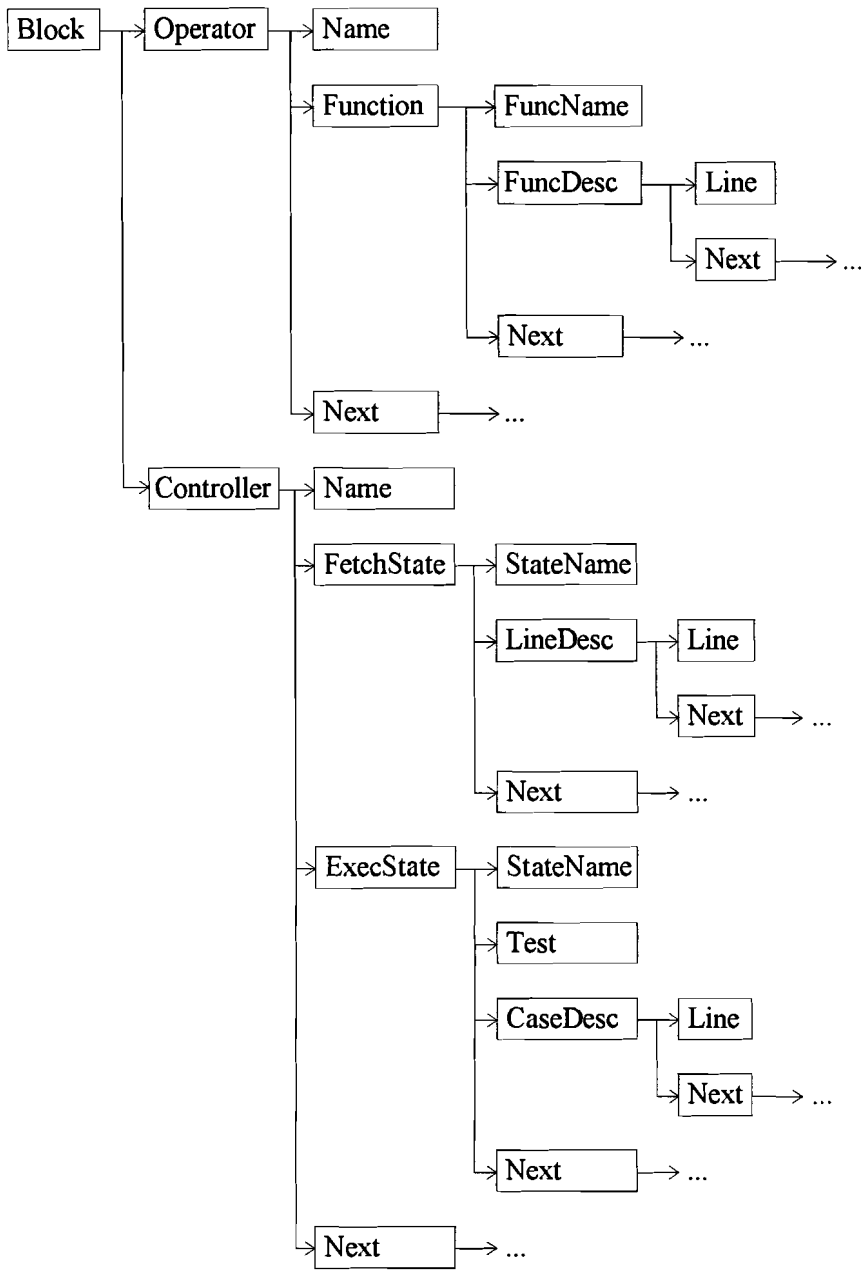

Figure 21: Data structure for implementation information storage

structur
e if at least the number of data buses and the interconnections between the operators and the buses are stored in it. This is only possible if the data structure, defined during the previous stage, is duitable for storing this kinds of information.
A procedure for going through the parse tree and storing the required information in the structure automatically has not been worked out yet.

Implementation information retrieval

This is the part of the translation program where the actual instruction set is implemented. The

translation tool will have to be programmed so that every time instruction set dependent graphical information for resources must be written to the IDaSS file, it will be retrieved from the data structure immediately. When all SimDes defined resources are placed in the schematic the special instruction set implementation blocks, operators and controllers, will be added. All graphical information and function description lines can be fetched from the data structure simply by copying the textual fields from one data structure block at a time to the IDaSS file.
To demystify this information retrieval stage a bit, an example of a procedure belonging to the data type in figure 21 is now presented. Suppose the data type is defined as 'cBlock' and a variable of this type called 'Block'. The variable contains information about the complete instruction set dependent part of the IDaSS processor design. One specific part of 'Block' has the following contents:

```
Operator = { "ALU" ,
             { "add" ,
               { "O = i1 + i2" ,
                 NULL
               } ,
               NULL
             } ,
             NULL
           }
```

This is a description of an ALU operator with one function: 'add'. The procedure to translate this kind of data into an IDaSS block would look like:

```
pOperator TempOper;
pFunction TempFunc;
PLineDesc TempLine;
pFetchState TempFetch;
pExecState TempExec;
int i;

TempOper = Block->Operator;
while (TempOper != NULL)
 {
  fprintf(idassfile,"#Operator %s \n/P8@58 18@14 \n",TempOper->Name);
  fprintf(idassfile,"IIi1 1 -3W%d \n",buswidth);
  fprintf(idassfile,"/P16@2 -4@0 \nIIi2 1 -3W%d \n",buswidth);
  fprintf(idassfile,"/P16@10 -4@0 \nOOo 1 -3W%d \n",buswidth);
  fprintf(idassfile,"/P0@10 2@0 \n");
  TempFunc = Block->Operator->Function;
  while (TempFunc != NULL)
   {
    fprintf(idassfile,"%s",TempFunc->FuncName);
    TempLine = TempFunc->FuncDesc;
    while (TempLine != NULL)
     {
      fprintf(idassfile,"%s",TempLine->Line);
      TempLine = TempLine->Next;
     }
    TempFunc = TempFunc->Next;
   }
  fprintf(idassfile,"S0 0 \n.Operator %s \n", TempOper->Name);
  TempOper = TempOper->Next;
 }
```

This example can only be used in case of a standard layout model. In the other cases additional lines must be included in the procedure, for retrieving the graphical information from the data structure.

The task of the translation tool programmer is to complete the translation process by going through the three stages of this method.

5.4 Remarks concerning specific implementation aspects

SimDes slice operator

In sub-section 2.2.5 the slice operator was described. This option in SimDes may be very convenient, but the implementation of slice expressions makes the operator(s) yet more sizeable. If two statements must be transformed to functions for an operator and those functions are

```
A = B + C and
A = D + E
```

only one function needs to be defined for the operator. The state controller will see to it that the contents of the right resources are available at the operators inputs. If on the other hand the statements are described as

```
A = B { 0 .. 7 } + C { 0 .. 7 } and
A = D { 8 .. 15 } + E { 8 .. 15 }
```

then two different functions must be defined for this operator.
It may however be possible to include an extra operator in the schematic that takes care of all scheduling operations instead of leaving it to the other operators. If it is provided with a controller connector where information is offered about the lower and higher bit selectors this could be a powerful block.

Alias implementation

Another SimDes option may cause serious problems when it has to be translated to IDaSS. In SimDes there is a possibility to ease the access to specific parts of resources by using aliases. Since IDaSS does not support this option, a way to translate aliases must be found. An obvious method is to first transform an alias back to the original resource, added with a slice operator, and subsequently translate this just like a statement. The first translation step can be performed by a procedure that reads the original resource from an alias description table. The problems that will remain for the last step are the same as described in the previous sub-section.

Flag setting

A subject that has been mentioned before (see sub-section 5.1.2) is the flag setting implementation. Although the problem is layout model dependent a general perspective is not entirely redundant. The processor status register that contains the individual flags is an unusual resource. Its bits will be affected by almost every instruction execution. When a layout model is chosen which makes use of more than one operator part of the problem will be solved because this register has its own operator at its disposal. But, even now, if more than one flag bit has to be set, this can only be done sequentially.
A suggestion to answer this is to make a small change to SimDes. If the register containing the flag bits would be described separately from the rest in its own description block, the translation program can be programmed so that it treats the flag register differently. All flagsetting statements can be combined automatically in one single operator function per instruction. In this way all flags

are set within one step. The new structure part of the SimDes language would be:

```
|[ resources    :
   |[ memory     : ... ]|
   |[ PC         : ... ]|
   |[ registers  : ... ]|
   |[ ports      : ... ]|

   |[ PSW        : ... ]|

   |[ aliases    : ... ]|
]|
```

In the PSW (processor status word) block one register description can be placed. The individual flags may be defined as aliases, so that they can be used separately.

Now that the main bottlenecks of the instruction set implementation are explained and a number of methods are described to deal with this problems, the research on the implementation part of the translation is in a stadium that makes it possible to evaluate the results.

6. Conclusions and recommendations

6.1 First translation results

The first step of developing a translation tool which generates IDaSS files from SimDes descriptions was to write a program which takes care of adding graphical information to the resource definitions in SimDes. The intention was to generate IDaSS designs based on one standard layout model. SimIdass is a tool which complies with this demand. It succeeds in deriving an IDaSS processor design from the description.
The model exists of a global arrangement of the IDaSS building block groups. A one to one projection method is used to derive the building blocks from the resource definitions. Combining this projection with the standard model all resources in the description can be placed in the schematic.
To prepare the design for instruction set implementation special places are reserved for the blocks which represent this task. In this model the implementation specific block groups are one ALU and one state machine controller. The ALU operator is accompanied by several registers to buffer data before and after operation. The state controller takes care of assigning functions to the resources and the ALU.
Buses also are added to the design during the SimIdass execution to make the layout design complete.
Some of the resources are translated in an ostensibly non-trivial way. The most important of them is the memory-block. The SimDes memory block is not meant to represent an on-chip memory resource. Therefore this block will be translated by SimIdass into a separate schematic, externally connected to the processor. A similar operation concerns the ports in the resource description.

6.2 Implementation methods

The SimIdass layout model has been prepared for implementing an instruction set, by reserving space in the schematic for an operator and a controller. But if the instruction set implementation is mapped on the SimIdass layout model, serious problems can be come upon. After locating the possible bottlenecks in the implementation process alternative layout models have been searched for. The three design alternatives are presented here:

Standard layout model

First a standard model of the resource layout is defined. Subsequently the translation tool must be programmed so that it can provide each resource of the required graphical information to place it in the IDaSS schematic. Before adding the two instruction set representing blocks, the functions for the operator and the state descriptions for the controller will have to be derived from the description. Finally the schematic will be completed with buses.

Advantages:

* The layout generation part of the translation tool is relatively simple.
* The IDaSS schematic is very orderly.

Disadvantages:

* The number of statement functions defined within the ALU operator can be very high.
* Execution time per instruction is relatively high, because all statements must be executed sequentially. Flag setting can not be performed in parallel with the rest of the instruction statements and will be done one by one.

Automatic data path synthesis

To avoid all main problems of the first implementation method another method can be used. The automatic synthesis of the complete data path always results in a design where the data flow is optimized as far as possible.

Advantage:

* Due to optimization the execution time will be reduced by comparison with the standard layout method.

Disadvantages:

* The programming of an automatic data path synthesis tool is very complex.
* The translation may result in a very chaotic schematic; in that case the simulation cannot be done orderly.

Distributed operator model

To deal with at least some of the problems of the first model a third method has been invented. It has the advantage of the simple layout generation due to a more or less standard layout structure, but the ALU functions will be distributed over several smaller operators. Each resource has its own operator and can communicate over more than one databus if necessary.

Advantages:

* Smaller operators.
* Several buses to optimize parallel execution of statements.
* Flags can be set in parallel with the other statements.
* Relatively orderly layout schematics.

Disadvantages:

* The layout is partially dependent on the instruction set.
* Flag setting will still be done sequentially: one flag at a time.
* Although a schematic will be quite orderly, it can become elongated. This because all resources will be placed in one row. If not, the routing of the buses causes an additional problem.

6.3 Recommendations for further development

It is up to the implementation tool programmer to deliberate over the different methods that can be used. If the period of time that is available for programming is short only the standard layout method is eligible. Otherwise the other two options deserve a further research especially the

distributed operator method, because the extra time needed will probably not balance the advantages it brings by comparison with the standard model.
If the programmer decides to use the SimIdass tool for further development, the implementation process described in section 5.3 can be used. Also the change to the SimDes language to allow separate flag handling must be considered.
The last remarks concern the future development of the source and target descriptions. If the SimDes description structure will be altered in future, for example to permit descriptions for parallel processor designs, the translation tool must be adjusted to this new grammar. The new version of the target description, IDaSS, will be out very soon. Although there will be some major changes the new IDaSS tool is able to read files written by or for the old version.

A Bibliography

References:

[Miya84] Miyagawa, A. et al
Cross-program simulator for any microprocessors
Conference: *Proceedings of the International Conference on Data Engineering* (1984)
Los Angeles, CA, USA, 24-27 April 1984
pag. 576-583

[Torr88] Torrelas, J. et al
Introductory user's guide to the architect's workbench tools
Technical Report CSL-TR-88-355, Stanford University
Computer Science Laboratory, May 1988

[Koom90] Koomen, C.J.
The design of communicating systems
Kluwer Academic Publishers, 1991 Dordrecht

[Vers90] Verschueren, A.C.
Interactive Design and Simulation System for Ultra Large Scale Integration
IDaSS manual, Eindhoven University of Technology
Digital Systems Group, July 1990

[Takk92] Takken, R.
Sequential Instruction Set Simulator Generator
Master Thesis Report EB 408, Eindhoven University of Technology
Digital Systems Group, October 1992

[Vers93] Verschueren, A.C.
The IDaSS file formats
IDaSS manual, Eindhoven University of Technology
Digital Systems Group, September 1993

[Diet75] Dietmeyer, D. et al.
Register transfer languages and their translation
Conference: *Digital system design automation* (1975)
pag. 117-218

[Beik90] Beikzadeh
A primary planner for high-level architectural synthesis
Conference: *Proceedings of the 33th midwest symposium on circuits and systems* (1990)
New York, NY, USA, 12-14 Augustus 1990
vol. 2, pag. 993-996

B SimDes Grammar and Production rules

The SimDes grammar can be characterized using the following items:

1 Start symbol
2 Non-terminals
3 Terminals
4 Production rules

1 Start symbol

Start symbol : description

2 Non-terminals

Non-terminals in order of occurrence:

```
description
resources
memory
pc
registers
register_declarations
register_declaration
ports
port_declarations
port_declaration
aliases
alias_declarations
alias_declaration
statistics
statistic_declarations
statistic_declaration
identifiers
rest_identifiers
designator1
initial_values
integer_list
rest_integer_list
implementations
rest_implementations
local_resources
fetchbuffer
function_block
functions
function
parameters
rest_parameters
parameter
return_type
fetch_block
instruction_block
instructions
instruction
opcode
rest_opcode
operand
rest_operand
traphandler_block
locals
local
statement_list
```

```
block
statement
alternatives
function_arguments
rest_arguments
designator2
```

3 Terminals

Terminal symbols which occur in the production rules are handed to the parser by the scanner. These symbols are placed within quotes, except for the Ident and Int symbols.
Terminals:

```
description
resources
memory
PC
ports
registers
aliases
implementation
functions
instructions
traphandler
statistics
fetch
fetchbuffer

if
then
else
for
to
downto
do
while
repeat
until
case
default
return
void

Ident
Int
```

4 Production rules

The SimDes grammar is a context-free grammar. A non-terminal is replaced by (n)one or more non-terminals or terminals. All rules consist of a non-terminal on the left-hand side of the colon and an optional non-terminal/terminal mix on the right-hand side.

```
non-terminal            : (non-terminal | terminal)*
```

Production rules:

```
/* global description production rules */

description             :
description             : |[ 'description' : Ident      resources
                                             statistics   implementations
]|


/* production rules resources */


resources               : |[ 'resources' : memory     pc
                                           registers  ports
                                           aliases            ]|

memory                  : |[ 'memory' : Ident [ Int ] { Int } ]|

pc                      : |[ 'PC' : Ident { Int } = { Int } ]|

registers               :
registers               : |[ 'registers' : register_declarations ]|
register_declarations   : register_declaration
register_declarations   : register_declaration ; register_declarations
register_declaration    :
register_declaration    : identifiers designator1 { Int } initial_values

ports                   :
ports                   : |[ 'ports' : port_declarations ]|
port_declarations       : port_declaration
port_declarations       : port_declaration ; port_declarations
port_declaration        :
port_declaration        : identifiers designator1 { Int }

aliases                 :
aliases                 : |[ 'aliases' : alias_declarations ]|
alias_declarations      : alias_declaration
alias_declarations      : alias_declaration ; alias_declarations
```

```
alias_declaration       :
alias_declaration       : Ident = Ident designator1 { Int }
alias_declaration       : Ident = Ident designator1 { Int .. Int }

identifiers             : Ident rest_identifiers
rest_identifiers        :
rest_identifiers        : , Ident rest_identifiers

designator1             : [ Int ] designator1
designator1             :

initial_values          : = { integer_list }
initial_values          :

integer_list            : Int rest_integer_list
rest_integer_list       :
rest_integer_list       : , Int rest_integer_list


/* statistics global to description */


statistics              :
statistics              : |[ 'statistics' : statistic_declarations ]|
statistic_declarations  : statistic_declaration
statistic_declarations  : statistic_declaration ; statistic_declarations
statistic_declaration   :
statistic_declaration   : identifiers designator1 { Int }


/* implementation block production rules */


implementations         : implementation rest_implementations
rest_implementations    :
rest_implementations    : implementation rest_implementations

implementation          : |[ 'implementation' : Ident
                                                local_resources
                                                statistics
                                                function_block
                                                fetch_block
                                                instruction_block
                                                traphandler_block ]|
```

```
/* resources local to implementation */

local_resources        : |[ 'resources' : fetchbuffer registers
                                          aliases                  ]|

fetchbuffer            : |[ 'fetchbuffer' : Ident [ Int ] { Int } ]|

/* functions local to implementation */

function_block         :
function_block         : |[ 'functions' : functions ]|

functions              :
functions              : function functions
function               : Ident ( parameters ) : return_type
                             { locals | statement_list }

parameters             :
parameters             : parameter rest_parameters
rest_parameters        :
rest_parameters        : , parameter rest_parameters
parameter              : identifiers { Int }

return_type            : { Int }
return_type            : void

/* fetch_block local to implementation */

fetch_block            : |[ 'fetch' : { locals | statement_list } ]|

/* instructions local to implementation */

instruction_block      : |[ 'instructions' : instructions ]|
instructions           :
instructions           : instruction instructions
instruction            : Ident ( opcode |  operand )
                             { locals | statement_list }
instruction            : Ident ( opcode )
                             { locals | statement_list }
```

```
opcode                  : Int rest_opcode
rest_opcode             :
rest_opcode             : , Int rest_opcode

operand                 :
operand                 : Ident { Int } rest_operand
operand                 : Ident { Int .. Int } rest_operand
rest_operand            :
rest_operand            : , Ident { Int } rest_operand
rest_operand            : , Ident { Int .. Int } rest_operand


/* traphandler local to implementation */


traphandler_block       :
traphandler_block       : |[ 'traphandler' : ]|
traphandler_block       : |[ 'traphandler' :
                                { locals | statement_list } ]|


/* locals used to declare locals preceding statement blocks */


locals                  : local ; locals
locals                  : local
local                   :
local                   : identifiers designator1 { Int }


/* From statement_list to statement */


statement_list          : block ; statement_list
statement_list          : block

block                   : { statement_list }
block                   : statement

statement               :
statement               : 'if' expression 'then' block
statement               : 'if' expression 'then' block 'else' block
statement               : 'for' Ident designator2 = expression
                                        'to' expression 'do' block
statement               : 'for' Ident designator2 = expression
                                   'downto' expression 'do' block
statement               : 'while' expression 'do' block
statement               : 'repeat block 'until' expression
```

```
statement               : 'case' expression
                                { alternatives  'default' : block }
statement               : 'return'
statement               : 'return' ( expression )
statement               : Ident ( function_arguments )
statement               : var   = expression
statement               : var  *= expression
statement               : var  /= expression
statement               : var  %= expression
statement               : var  += expression
statement               : var  -= expression
statement               : var  &= expression
statement               : var  ^= expression
statement               : var  |= expression
statement               : var <<= expression
statement               : var >>= expression
statement               : var  ++
statement               : var  --

alternatives            :
alternatives            : Int : block alternatives

function_arguments      :
function_arguments      : expression rest_arguments
rest_arguments          :
rest_arguments          : , expression rest_arguments

designator2             :
designator2             : [ expression ] designator2

var                     : Ident designator2
var                     : Ident designator2 { expression }
var                     : Ident designator2 { expression .. expression }


/* expressions used in statements */


expression              : ( expression )
expression              : Int
expression              : Ident designator2
expression              : Ident ( function_arguments )


/* ternary operator { slice } */


expression              : expression { expression .. expression }
```

```
/* binary operators */


expression              : expression *  expression
expression              : expression /  expression
expression              : expression %  expression
expression              : expression +  expression
expression              : expression -  expression
expression              : expression &  expression
expression              : expression ^  expression
expression              : expression |  expression
expression              : expression == expression
expression              : expression != expression
expression              : expression <  expression
expression              : expression >  expression
expression              : expression <= expression
expression              : expression >= expression
expression              : expression << expression
expression              : expression >> expression


/* unary operator */

expression              : ~expression
```

C Parse tree data type

In the "TreeType.h" file the data type of the parse tree is defined. The part of this file that contains the actual definition is printed here:

```
/* ------------------------------------------------------------ */
/*          Parse-tree datastructure definition                 */
/* ------------------------------------------------------------ */


/* ----Description-node------------------------------------- */

struct hDescription    { tIdent                 Ident;
                         struct hResource      *Resource;
                         struct hDecl          *Statistic;
                         struct hImplementation *Implementation;
                         tPosition              Position;      };


/* ----Resource-node---------------------------------------- */

struct hResource       { struct hDecl          *Memory;
                         struct hDecl          *Pc;
                         struct hDecl          *Register;
                         struct hDecl          *Port;
                         struct hAlias         *Alias;       };

struct hDecl           { struct hIdList        *Ident;
                         struct hIntList       *Array;
                         Integer                Width;
                         struct hIntList       *Init;
                         struct hDecl          *Next;
                         tPosition              Position;      };

struct hAlias          { tIdent                 Ident;
                         tIdent                 From;
                         struct hIntList       *Array;
                         Integer                Offset;
                         Integer                Type;
                         struct hAlias         *Next;
                         tPosition              Position;      };

struct hIdList         { tIdent                 Ident;
                         struct hIdList        *Next;
                         tPosition              Position;      };

struct hIntList        { Integer                Int;
                         struct hIntList       *Next;
                         tPosition              Position;      };

struct hExprList       { struct hExpr          *Expr;
                         struct hExprList      *Next;
                         tPosition              Position;      };


/* ----Implementation-node---------------------------------- */

struct hImplementation { tIdent                 Ident;
                         struct hImResource    *Resource;
                         struct hDecl          *Statistic;
                         struct hImFunction    *Function;
```

```
                        struct hImFetch         *Fetch;
                        struct hImInstruction   *Instruction;
                        struct hImTrapHandler   *TrapHandler;
                        struct hImplementation  *Next;
                        tPosition                Position;      };

struct hImFetch         { struct hDecl            *Local;
                        struct hStat            *Stat;         };

struct hImResource      { struct hDecl            *Register;
                        struct hAlias           *Alias;
                        struct hDecl            *FetchBuffer;  };

struct hImFunction      { tIdent                   Ident;
                        struct hDecl            *Param;
                        struct hReturn          *Return;
                        struct hDecl            *Local;
                        struct hStat            *Stat;
                        struct hImFunction      *Next;
                        tPosition                Position;      };

struct hReturn          { Integer                  Type;
                        tPosition                Position;      };

struct hImInstruction   { tIdent                   Ident;
                        struct hIntList         *Opcode;
                        struct hOperand         *Operand;
                        struct hDecl            *Local;
                        struct hStat            *Stat;
                        struct hImInstruction   *Next;
                        tPosition                Position;      };

struct hOperand         { tIdent                   Ident;
                        Integer                  Start;
                        Integer                  End;
                        struct hOperand         *Next;
                        tPosition                Position;      };

struct hImTrapHandler   { struct hDecl            *Local;
                        struct hStat            *Stat;         };


/* ----Statement-node---------------------------------------- */


struct hIf              { struct hExpr            *Cond;
                        struct hStat            *Then;
                        struct hStat            *Else;         };

struct hFor             { struct hVar             *Ident;
                        struct hExpr            *Start;
                        struct hExpr            *End;
                        struct hStat            *Stat;         };

struct hWhile           { struct hExpr            *Expr;
                        struct hStat            *Stat;         };

struct hRepeat          { struct hExpr            *Expr;
                        struct hStat            *Stat;         };

struct hCase            { struct hExpr            *Expr;
                        struct hAlt             *Alt;
                        struct hStat            *Default;      };

struct hAlt             { Integer                  Cond;
                        struct hStat            *Stat;
```

```
                        struct hAlt             *Next;
                        tPosition               Position;     };

struct hReturnStat    { struct hExpr            *Return;      };

struct hFuncRef       { tIdent                  Ident;
                        struct hExprList        *Expr;        };

struct hAssign        { struct hVar             *Var;
                        struct hExpr            *Expr;        };

struct hVar           { tIdent                  Ident;
                        struct hExprList        *Array;
                        struct hExpr            *Start;
                        struct hExpr            *End;
                        tPosition               Position;     };

union uStat           { struct hIf              If;
                        struct hFor             For;
                        struct hWhile           While;
                        struct hRepeat          Repeat;
                        struct hCase            Case;
                        struct hFuncRef         FuncRef;
                        struct hAssign          Assign;
                        struct hReturnStat      Return        };

struct hStat          { union uStat             Stat;
                        Integer                 Kind;
                        struct hStat            *Next;
                        tPosition               Position;     };

/* ----Expression-node------------------------------------ */

struct hIdent         { tIdent                  Ident;
                        struct hExprList        *Array;       };

struct hInteger       { Integer                 Int;          };

struct hUnary         { struct hExpr            *Expr;        };

struct hBinary        { struct hExpr            *Left;
                        struct hExpr            *Right;       };

struct hSlice         { struct hExpr            *Expr;
                        struct hExpr            *Start;
                        struct hExpr            *End;         };

union  uExpr          { struct hSlice           Slice;
                        struct hIdent           Ident;
                        struct hInteger         Int;
                        struct hFuncRef         FuncRef;
                        struct hUnary           Unary;
                        struct hBinary          Binary;       };

struct hExpr          { union uExpr             Expr;
                        Integer                 Kind;
                        Integer                 Type;
                        tPosition               Position;     };
```

```
/* ----Define-pointers-to-nodes----------------------------- */

typedef struct hDescription     *pDescription;
typedef struct hResource        *pResource;
typedef struct hDecl            *pDecl;
typedef struct hAlias           *pAlias;
typedef struct hIdList          *pIdList;
typedef struct hIntList         *pIntList;
typedef struct hExprList        *pExprList;
typedef struct hImplementation  *pImplementation;
typedef struct hImResource      *pImResource;
typedef struct hImFunction      *pImFunction;
typedef struct hReturn          *pReturn;
typedef struct hImFetch         *pImFetch;
typedef struct hImInstruction   *pImInstruction;
typedef struct hOperand         *pOperand;
typedef struct hImTrapHandler   *pImTrapHandler;
typedef struct hAlt             *pAlt;
typedef struct hVar             *pVar;
typedef struct hStat            *pStat;
typedef struct hExpr            *pExpr;
typedef struct hDesignator      *pDesignator;
```

D ALUout databus definition

```
void IdassBus (pResource Resource, pImResource ImResource, int buswidth,
               int lotype, int c0, int c1)
{
  char * name;
  char * cout;
  char * cin;
  int i, can, os, no, cp, line, sp;
  pDecl TempReg, TempPort;
  pIdList TempName;
  pIntList Array;

  name = (char *)calloc(30,sizeof(char));

  /* ---------------------- ALU out bus --------------------- */

        /* ================= connectors ================== */

  fprintf(idassfile,"#Bus aluoutbus \nC1 ALU o \nC2 ALUREG *i \nC3 TEMP *i \n");
  if (lotype > 3)
    fprintf(idassfile,"C4 DtoM i \n");
  else
    fprintf(idassfile,"C4 d_write d_write \n");
  if (lotype%2 == 0)
    fprintf(idassfile,"C5 DtoA i \n");
  else
    fprintf(idassfile,"C5 AREG *i \n");
  if ((lotype > 1) && (lotype < 6))
    fprintf(idassfile,"C6 PC *i \n");
  else
    fprintf(idassfile,"C6 DtoPC i \n");
  can = 7;
  TempReg = Resource->Register;
  while (TempReg != NULL)
   {
    if (TempReg->Width != buswidth)
     {
      TempName = TempReg->Ident;
      while (TempName != NULL)
       {
        GetString (TempName->Ident,name);
        fprintf(idassfile,"C%d Dto%s i \n",can++,name);
        TempName = TempName->Next;
       }
     }
    TempReg = TempReg->Next;
   }
  TempReg = ImResource->Register;
  while (TempReg != NULL)
   {
    if (TempReg->Width != buswidth)
     {
      TempName = TempReg->Ident;
      while (TempName != NULL)
       {
        GetString (TempName->Ident,name);
        fprintf(idassfile,"C%d Dto%s i \n",can++,name);
        TempName = TempName->Next;
       }
     }
    TempReg = TempReg->Next;
   }
  TempReg = Resource->Register;
  while (TempReg != NULL)
   {
    if (TempReg->Width == buswidth)
     {
      TempName = TempReg->Ident;
      while (TempName != NULL)
```

```
     {
      if (TempReg->Array == NULL)
       {
        cin = "*i";
       }
      else
       {
        cin = "i";
       }
      GetString (TempName->Ident,name);
      fprintf(idassfile,"C%d %s %s \n",can++,name,cin);
      TempName = TempName->Next;
     }
   }
  TempReg = TempReg->Next;
 }
TempReg = ImResource->Register;
while (TempReg != NULL)
 {
  if (TempReg->Width == buswidth)
   {
    TempName = TempReg->Ident;
    while (TempName != NULL)
     {
      if (TempReg->Array == NULL)
       {
        cin = "*i";
       }
      else
       {
        cin = "i";
       }
      GetString (TempName->Ident,name);
      fprintf(idassfile,"C%d %s %s \n",can++,name,cin);
      TempName = TempName->Next;
     }
   }
  TempReg = TempReg->Next;
 }

TempPort = Resource->Port;
while (TempPort != NULL)
 {
  TempName = TempPort->Ident;
  while (TempName != NULL)
   {
    GetString(TempName->Ident,name);
    if (TempPort->Array == NULL)
      fprintf(idassfile,"C%d %sb w \n",can++,name);
    else
      IndexBus (name, TempPort->Array, &can, 'w');
    TempName = TempName->Next;
   }
  TempPort = TempPort->Next;
 }

       /* ================== nodes =================== */

fprintf(idassfile,"/N%d 5@85 \n/N%d 5@77 \n/N%d 5@69 \n/N%d 5@55 \n",can++,can++,can++,can++);
if (lotype > 3)
 {
  fprintf(idassfile,"/N%d 49@55 \n",can++);
  fprintf(idassfile,"/N%d 49@21 \n",can++);
  fprintf(idassfile,"/N%d 49@19 \n",can++);
 }
else
 {
  fprintf(idassfile,"/N%d 29@55 \n",can++);
  fprintf(idassfile,"/N%d 29@21 \n",can++);
  fprintf(idassfile,"/N%d 29@19 \n",can++);
 }
os = 0;
if (c0 > 0)
 {
```

```
  fprintf(idassfile,"/N%d 157@5 \n/N%d 157@19 \n",can++,can++);
  for (i=0; i<c0+c1; i ++)
    fprintf(idassfile,"/N%d 157@%d \n",can++,29+8*i);
 }
else if ((lotype < 2) || (lotype > 5))
 {
  fprintf(idassfile,"/N%d 157@5 \n/N%d 157@19 \n",can++,can++);
  for (i=0; i<c1; i ++)
    fprintf(idassfile,"/N%d 93@%d \n",can++,29+8*i);
  fprintf(idassfile,"/N%d 93@19 \n",can++);
  os = 1;
 }
else
 {
  fprintf(idassfile,"/N%d 133@5 \n/N%d 133@19 \n",can++,can++);
  for (i=0; i<c1; i ++)
    fprintf(idassfile,"/N%d 93@%d \n",can++,29+8*i);
  fprintf(idassfile,"/N%d 93@19 \n",can++);
  os = 1;
 }

line = (c0+c1) < 10 ? 109 : 29+8*(c0+c1);
cp = 0;
TempPort = Resource->Port;
while (TempPort != NULL)
 {
  TempName = TempPort->Ident;
  while (TempName != NULL)
   {
    sp = 1;
    Array = TempPort->Array;
    while (Array != NULL)
     {
      sp *= (Array->Int);
      Array = Array->Next;
     }
    for (i=0; i<sp; i++)
     {
      fprintf(idassfile,"/N%d 5@%d \n",can++,line+12*i);
      cp ++;
     }
    TempName = TempName->Next;
   }
  TempPort = TempPort->Next;
 }

if (lotype%2 == 0)
  fprintf(idassfile,"/N%d 73@13 \n/N%d 73@19 \n",can++,can);
else if (lotype > 3)
  fprintf(idassfile,"/N%d 49@13 \n",can);
else
  fprintf(idassfile,"/N%d 49@13 \n/N%d 49@19 \n",can++,can);

      /* ================= segments ================ */

no = 6 + c0 + c1 + cp;
fprintf(idassfile,"/S4 %d \n/S5 %d \n/S6 %d \n",no+6,2*no+4+os,no+8);
fprintf(idassfile,"/S%d 3 \n/S%d 2 \n/S%d %d \n",no+1,no+2,no+2,no+1);
fprintf(idassfile,"/S%d 1 \n/S%d %d \n/S%d %d \n",no+3,no+3,no+2,no+4,no+3);
fprintf(idassfile,"/S%d %d \n/S%d %d \n/S%d %d \n",no+4,no+5,no+6,no+5,no+7,no+6);
fprintf(idassfile,"/S%d %d \n",no+8,no+9);
for (i=7; i<7+c0+c1; i++)
  fprintf(idassfile,"/S%d %d \n",i,no+3+i);
for (i=no+10; i<no+9+c0+c1; i++)
  fprintf(idassfile,"/S%d %d \n",i,i+1);
if (c0 > 0)
 {
  fprintf(idassfile,"/S%d %d \n",no+9,no+10);
  if ((lotype%2 != 0) && (lotype > 3))
   {
    fprintf(idassfile,"/S%d %d \n",2*no+4+os,no+7);
    fprintf(idassfile,"/S%d %d \n",no+7,no+9);
   }
  else
```

```
    {
     fprintf(idassfile,"/S%d %d \n",2*no+4+os,2*no+5+os);
     fprintf(idassfile,"/S%d %d \n",no+7,2*no+5+os);
     fprintf(idassfile,"/S%d %d \n",2*no+5+os,no+9);
    }
  }
 else
  {
   fprintf(idassfile,"/S%d %d \n",no+10+c1,no+9);
   fprintf(idassfile,"/S%d %d \n",no+10+c1,no+10);
   if ((lotype%2 != 0) && (lotype > 3))
    {
     fprintf(idassfile,"/S%d %d \n",2*no+4+os,no+7);
     fprintf(idassfile,"/S%d %d \n",no+7,no+10+c1);
    }
   else
    {
     fprintf(idassfile,"/S%d %d \n",2*no+4+os,2*no+5+os);
     fprintf(idassfile,"/S%d %d \n",no+7,2*no+5+os);
     fprintf(idassfile,"/S%d %d \n",2*no+5+os,no+10+c1);
    }
  }
 if (cp>0)
  {
   fprintf(idassfile,"/S%d %d \n",no+1,no+10+c0+c1+os);
   for (i=7+c0+c1; i<7+c0+c1+cp; i++)
     fprintf(idassfile,"/S%d %d \n",i+9+c0+c1+cp+os,i);
  }
 for (i=1; i<cp; i++)
   fprintf(idassfile,"/S%d %d \n",no+9+c0+c1+os+i,no+10+c0+c1+os+i);
 fprintf(idassfile,".Bus aluoutbus \n");
}
```

E 'WriteIdass' function descriptions

The functions that together form the layout generation tool are briefly explained below. They are all joint together in the "WriteIdass.c" file.

```
void IdassDescription (pDescription Description,
                       pImplementation Implementation)
```

This is the main function for translation of a Description (stored in a parse tree) into an IDaSS file. It starts with describing the top-level in IDaSS and subsequently uses all other functions to add blocks and buses.

```
void IdassMemory (pDecl Memory)
```

The IdassMemory function is used to describe the Memory schematic containing the RAM block in IDaSS.

```
void IdassPortTree (pDecl Port, int * countadr)
```

Here the external registers are added to the top-level representing the ports. It makes use of the IndexPort and the IdassPort functions where the actual IDaSS description lines are produced.

```
void IndexPort (char * name, int width, int * lineadr, pIntList TempIndex,
                int * countadr)
```

A special function between IdassPortTree and IdassPort is created to deal with definitions of arrays of ports.

```
void IdassPort (char * name, int line, int width)
```

This function takes care of the production of the IDaSS description lines for the external port registers.

```
void IdassResource (pDescription Description,
                    pImplementation Implementation, int counter)
```

Here the contents of the processor block are filled in. IdassResource uses many other functions each of which describes a specific part of the processor.

```
void IdassSheetSize (pDescription Description,
                     pImplementation Implementation)
```

A size for the IDaSS worksheet must be specified. At this time standard values are used, but later on this can be replaced by a procedure that calculates the required format from the description.

```
void GetMax (pResource Resource, pImResource ImResource, int * bwadr,
             int * typeadr)
```

This very important procedure searches for the largest bitwidth of all resources in the processor description and assigns the accompanying value to the variable "lotype".

```
void IdassConnect (pDecl Port)
void IndexConnect (char * name, pIntList Array, int * lineadr)
```

In the top-level schematic superconnectors for the port connections must be added to the processor block. This is done by the combination of the IdassConnect and the IndexConnect function.

```
void IdassMemCon (pDecl Memory, int buswidth, int lotype)
```

This function adds the superconnectors for the memory to the processor block at the top-level.

```
void IdassPc (pDecl PC, int buswidth, int lotype, int bits)
```

Here the programcounter, PC-buffers and bitwidth operator are placed in the processor schematic.

```
void IdassFb (pDecl FB)
```

IdassFb takes care of writing the fetchbuffer schematic.

```
void IdassRegTree (pDecl Register, int * lineadr, int equals,
                   int * countadr, int buswidth)
void IndexReg (char * name, int width, int column, int * lineadr,
               pIntList TempIndex, pIntList * TempValadr)
void IdassRegister (char * name, int column, int line, int width,
                    int value, char * ts1, char * ts2)
```

This combination of functions writes the register information to the IDaSS file, completed with bitwidth adjustment operators where necessary.

```
void IdassPortCon (pDecl Port, int * lineadr, int buswidth)
void IndexPortCon (char * name, pIntList Array, int * lineadr,
                   int width, int buswidth)
```

The superconnectors for the ports are described for the processor schematic, and so are the operators which determine the direction of communication.

```
void IdassAlu (pDescription Description,
               pImplementation Implementation, int buswidth, pBlock Block)
```

The ALU operator and all additional registers are described here. The ALU functions are retrieved from the information in the variable Block.

```
void IdassBus (pResource Resource, pImResource ImResource, int buswidth,
               int lotype, int c0, int c1)
void IndexBus (char * name, pIntList Array, int * canadr, char rw)
void IndexPortBus (char * name, pIntList Array, int * lineadr)
```

All bus descriptions are written to the IDaSS design file.

```
void IdassEnvBus (pDescription Description)
void IndexEnvBus (char * name, char * prtname, pIntList Array,
                  int * lineadr)
```

Last of all the external buses must be added to the schematic at the top-level. The processor block will be connected to the memory block and to the port-registers.

F Implementation information data structure

In the "BlockFunc.h" file the data type is defined in which all information regarding the instruction set implementation dependent blocks can be stored.

```
/* ----Block-node------------------------------------------ */

struct iBlock          { struct iOperator       *Operator;
                         struct iController     *Controller;    };

/* ----Operator-node--------------------------------------- */

struct iOperator       { char *                  Name;
                         struct iFunction       *Function;
                         struct iOperator       *Next;          };

struct iFunction       { char *                  FuncName;
                         struct iLineDesc       *FuncDesc;
                         struct iFunction       *Next;          };

struct iLineDesc       { char *                  Line;
                         struct iLineDesc       *Next;          };

/* ----Controller-node------------------------------------- */

struct iController     { char *                  Name;
                         struct iFetchState     *FetchState;
                         struct iExecState      *ExecState;
                         struct iController     *Next;          };

struct iFetchState     { char *                  StateName;
                         struct iLineDesc       *LineDesc;
                         struct iFetchState     *Next;          };

struct iExecState      { char *                  StateName;
                         char *                  Test;
                         struct iLineDesc       *CaseDesc;
                         struct iExecState      *Next;          };

/* ----Define-pointers-to-nodes---------------------------- */

typedef struct iBlock          * pBlock;
typedef struct iOperator       * pOperator;
typedef struct iFunction       * pFunction;
typedef struct iLineDesc       * pLineDesc;
typedef struct iController     * pController;
typedef struct iFetchState     * pFetchState;
typedef struct iExecState      * pExecState;
```