From IDea to IC : the higher levels of a silicon compiler

Stok, L.

Award date:
1986

From IDea to IC

Leon Stok
Eindhoven
August 1986

Eindhoven University of Technology
Department of Electrical Engineering
Automatic System Design group

From IDea to IC, the higher levels of a silicon compiler.

by L. Stok

Master thesis
reporting on graduation work
performed from 1.01.86 to 28.08.86
by order of prof. dr.-ing. J.A.G. Jess
and supervised by drs. R. v.d. Born
and ir. G.L.J.M. Janssen

Abstract

The main problem discussed in this report is: given an algorithm represented as a program in some high level language, how do we map that algorithm onto a hardware structure that implements the algorithm.

The approach presented here is the following: The algorithm is parsed and translated into a syntax tree. From this tree a special data flow graph, *the demand graph*, is made. On this graph several optimisations can be done to make the graph structure better realisable. Several inefficiencies introduced by the designer may also removed. Essential is that the optimisations transform a demand graph in a semantically equivalent demand graph.

Further the demand graph is compiled into a hardware structure. This hardware structure consists of a list of modules with their interconnections and a state machine. The compilation is done by generating alternative implementations, using a dynamic programming technique, and choosing the optimal implementation. This choice is made with the information provided by the module library, concerning the area, dissipation and speed of the modules.

The algorithms described in this report are coded in CommonLisp. The module list and state machine have automatically been generated for some algorithms.

CONTENTS

LIST OF FIGURES

Introduction.

This report is the result of my work done during my graduated period in the Automatic System Design Group (ES) of the department of Electrical Engineering at the Eindhoven University of Technology.

This group has several research projects concerning the development of tools for VLSI design. Some of these projects are contributions to the NELSIS/ICD (NEderlands ontwerpSysteem voor geIntegreerde Schakelingen / Integrated Circuit Design) project, which is a cooperation of the Dutch Universities of Technology and several companies in Great-Britain, Germany and the Netherlands.

The ESPRIT-991 project concerns Silicon Compilation. Silicon compilation is the automatic translation of a behavioural (algorithmic) description of a circuit into an implementable layout. Silicon compilation becomes increasingly important with the development of the IC technology. The technology enables to design very complex systems. These large systems cannot be designed by hand. Consequently, there will be a large market for silicon compilers in the near future.

At this place I would like to thank the group ES for the support given. Especially I would like to thank prof. J.A.G. Jess, who made this research project possible, and drs. R. v.d. Born and ir. G.L.J.M. Janssen for their useful discussions and continuous support. Furthermore I thank R. v.d. Born for proofreading this report and the suggestions he made for improvements.

Leon Stok

1. Hardware synthesis systems.

1.1 Introduction.

The continuing improvements in the integrated circuit technology have made possible to integrate increasingly complex circuits. The design of systems currently implementable on a single integrated circuit requires extensive use of design aids for such tasks as simulation and design verification. These tools typically aid in analysing a design once it has been specified. Missing at the systems level of design are those aids which help in creating or synthesising a design. The need for such design aids will grow because nowadays the complexity of the designs increases.

Although design synthesis was formerly considered to be the realm of the creative designer, automatic and semi-automatic programs are now being developed. As we move into the VLSI era, the demand for more capable system IC's requires even greater productivity at all levels of the design process. Thus, development of synthesis tools for the creative design process has become an important research area.

Synthesis is the creation of a detailed design from an abstract specification. Digital system design actually consists of many synthesis steps, each adding more detail.

Their use promises further benefits.

- *More design alternatives.* Designers can specify parts of the design and have the synthesis program fill in details quickly, or they can change constraint specifications so the synthesis aid specifies a different design.

- *Correctness by construction.* Human designers can make errors in the synthesis steps. When it is proved that a synthesis program correctly implements a specification, such design errors are avoided.

- *Multi level representations.* Synthesis programs can maintain correlations between abstract specifications and detailed design in the form of a representation with multiple levels of abstraction. The representation supports the use of powerful design aids such as mixed level simulators and timing verifiers.

Another advantage of automatic synthesis is the availability of IC technology also to the non expert designer, which offers not only economic advantages but also the possibility

of protecting know-how.

Automatic design systems may be particularly of use if instead of speed and/or area the main criteria are design costs, and especially design time. Design time for new circuits can be reduced to a few days. Special purpose chips to implement certain algorithms in silicon are applications well suited for this approach. Examples are network controllers, operating system functions, signal processing applications, special processors, etc. The applicability of silicon compilers will primarly be in the fabrication of circuits that do not stretch existing technology to its limits. For example: it will be very difficult for a silicon compiler to use the speed of the circuits to their limits. There always has to be a safety margin. On the other side a silicon compiler gives the designers the opportunity to use the advantages of the new technologies. The more abstract level of thinking about the design makes it possible to create more complex designs. The class of systems for which a silicon compiler can be used is large enough to merit further research.

1.2 System description.

The goal of our project is to develop a system synthesising a circuit from a high level description of a system. The high level description is a behavioural description. Usually the behaviour of a circuit is described using natural language. This description deals with the functions to be implemented and the requirements concerning power, reliability, pin-out, timing, technology etc. to be fulfilled. A formal description is nowadays often restricted to finite automata or function tables. Compared to context-free languages they do not allow a comfortable description of modular or hierarchical systems. We propose a more general approach by using a description of the algorithm in a context-free language similar to common programming languages. This high-level-description is given in a language like Pascal, C or LISP.

A *silicon compiler* is a set of tools able to transform such a description into a realisable layout. First we present globally what a *silicon compiler* does. We will describe a relation between the algorithm and the hardware.

1. The *processing unit* will take care of the variables, of the procedures and functions and of the assignments; intuitively the variables can be associated with registers and the function names will be assigned combinational logic circuits. Finally, the assignments will become functional register transfers

of the type

R:=F(R);

meaning that the contents of the set of registers R is to be loaded with a function F of the content of these registers.

2. The *control unit* will take care of the program itself i.e. of the constructs *while ... do*, *if ... then ...else*, *etc.*, of their sequencing and of the condition variables, i.e. of the binary variables providing the truth value of the conditions to be evaluated.

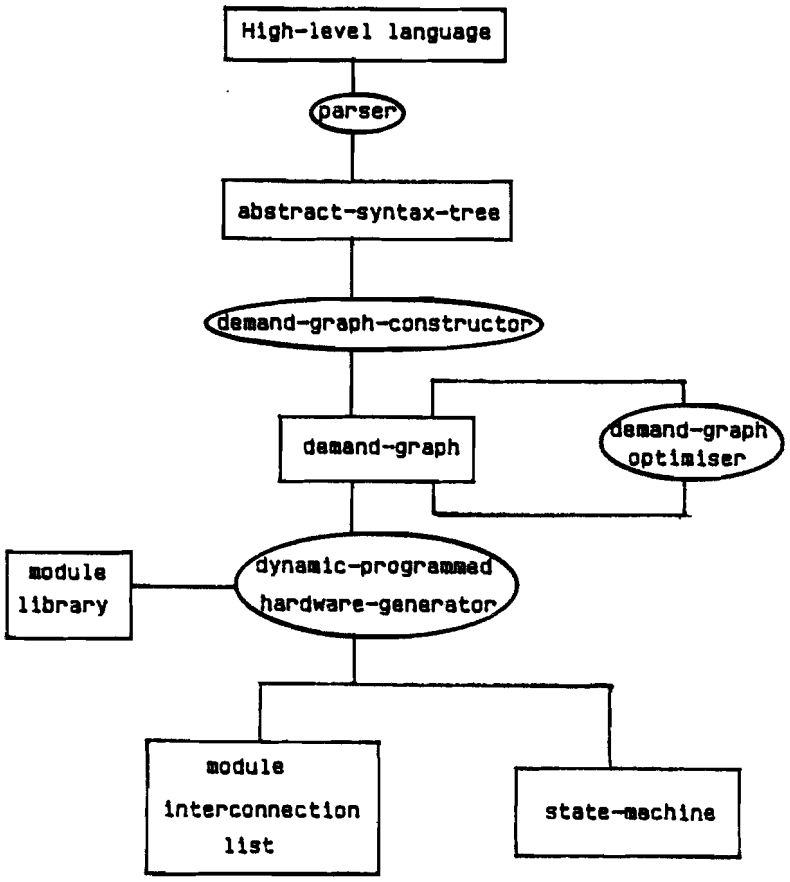


Figure 1.1. System overview.

The system is partitioned in several intermediate results and tools. The tools (shown in ellipses) convert the intermediate results (shown in boxes) to each other.
This partitioning of the system has several advantages: (see

fig. 1.1)

- The implementation of the system can be done in several steps.

- Between all stages we can display the intermediate results and make tools to interfere in these results. This can be useful when the design system is not fully automatic, and interaction with the designer is needed to synthesise a more optimal circuit.

- Libraries can be linked together into the system at several stages. This is important when complicated designs have to be made. We can use the results gathered in earlier designs. For example: we can make a procedure library at the language level and a library containing a set of demand graphs at the demand graph level.

- The *demand graph* can be translated into hardware by several hardware generators. We can use an expert system, an interactive system or a system that translates the whole *demand graph* at once, like our present-day system does.

This report describes the transformation from the *syntax tree* to the *demand graph* and from here to the symbolic hardware representation. These transformations are coded in CommonLisp during this project. Before going into detail in the following chapters we will shortly describe the components of the system.

1.2.1 *The high level language.*

"The symbol-making function is one of man's primary activities, like eating, looking, or moving about. It is the fundamental process of the mind, and goes on all the time."
S.K. Langer

"Man's achievements rest upon the use of symbols."
A. Korzybski

"Language ... makes progress possible."
S.I. Hayakawa

From "Language in Thought and Action" by S.I. Hayakawa, Harcourt, Brace and Company, 1949

As indicated by the quotations, languages give people the possibility to express and communicate their ideas. The

purpose of a design language is to permit efficient communication between the designer and the application design tools. But not only the communication with the machine is important. Nowadays designs are such complex that they cannot be made by one man. Thus some communication has to take place between the designers in the project team. The design language has to be suitable for this purpose too. The availability of application design tools to be used with a language is essential to the acceptance of the language by the design community.

There are several advantages when using a high level language and a high level silicon compiler:

1. Time consuming low level simulation and circuit verification are no longer needed when the system design is started from a high level.

2. The language gives the designers a communication and documentation medium. Formal description of a design is then possible.

3. The designers can think about their design at a more abstract level, therefore the time to develop a complex system is decreased considerably.

Once a design language is defined, it can serve as a basis for many design tools. But when defining a language we have to take care of supporting the following language features:

- Both human and machine readable functional specifications and documentation must be generated.

- Design management. The design data has to be subdivided into parts, conform to the way the designer thinks about the design.

- Behavioural descriptions. The algorithms, when expressed in the language, must reflect the designer thoughts about the algorithm. The designer has to be able to express in the language the way he thinks about the design.

- Description of a design's environment. The design has to fulfill certain specifications, as timing, signal levels and dissipation. Some special language constructs are needed to express these constraints put on the design by its environment.

- When the language is also used to serve the silicon compiler with more structural descriptions, it must be

possible to express a structural description and a timing description. It would be nice if the language is extensible.

A Behaviour Description Language (BDL) is used as the input to our silicon compiler. In this stage of our project we did neither develop a new language nor decided what existing language we could use. Instead we use the *syntax tree* of the language as input. The definition of the *syntax tree* is given in chapter 2. The *syntax tree* puts some constraints on the input language but there is a certain degree of freedom in choosing our language. This strategy has the advantage that we can add language structures during the project without the need to rewrite the parser each time. When all language elements are known the language can be defined or chosen. From this language a *syntax tree* is build using conventional compiler techniques [Aho86]. During the research described in this report the *syntax tree* is used as the input to the silicon compiler. Because we use a user friendly description of the *syntax tree* (see Appendix A), it does not raise too many difficulties to express an algorithm in the *syntax tree*.

1.2.2 *Demand graph constructor.*

The next intermediate result (see fig. 1.1) is the *demand graph*. The *demand graph* represents both data flow and control flow of the system described in the BDL. Nodes represent both the operations on the data and the direction in which the data flows. The edges represent the relation between a definition and a use of a variable. The role of the nodes and the edges will become clear in chapter 2.

The *demand graph* is, in a sense, independent from the specification given by the designer: different BDL specifications may lead to the same demand graph. So the graph does not directly represent the BDL description, but merely represents the intention the designer has put in the description.

Because of the nature of the data flow representation, the synthesis programs can change the order of operations specified in the high-level description - so long as data dependencies are satisfied - and can change design parallelism.

The tool which converts the *syntax tree* to the *demand graph* is the demand graph constructor. The constructor traverses the *syntax tree* and generates the appropriate nodes and edges of the *demand graph*.

1.2.3 *Demand graph optimisations.*

The *optimiser* converts a *demand graph* to a functionally equivalent *demand graph*. These conversions are done because they will result in a more efficient implementation of the algorithm. Certain optimisations are made to improve the description made by the designer. The designer can use some elements in his description to make the description more readable. For example the use of constants can make a description easier to read but will cause inefficiencies in the implementation. The *demand graph* is a useful representation for these optimisations. Most optimisations are similar to those used in optimising compilers. We will describe some optimisations here. The implemented optimisations are discussed in chapter 3. A survey of optimisations used in optimising compilers can be found in [Kenn81].
Some optimisations:

- *Redundant subexpression elimination.* If two operators that both compute the expression A * B are separated by code which contains no store into either A or B, then the second operator can be eliminated if the result of the first is saved.

- *Constant folding.* If all the inputs to an operator are constants whose values are known, the result of the operator can be computed at compile time and stored instead of the operator.

- *Code motion.* Operators that depend upon variables whose values do not change in a loop may be moved out of the loop, improving performance by reducing the operators 'frequency of execution.

- *Strength reduction.* Operators that depend on the loop induction variable cannot be moved out of the loop, but sometimes they can be replaced by less expensive operators.

- *Variable folding.* Statements of the form A:=B will become useless if B can be substituted for subsequent uses of A.

- *Dead code elimination.* If transformations like variable folding are successful, there will be many operators whose results are never used. Dead code elimination detects and deletes such operators.

- *Procedure integration.* Under certain circumstances, a procedure call can be replaced by the body of the

procedure being called.

Some other techniques from the optimising compilers can be used during the hardware generation. For example register allocation, scheduling of operations and detection of parallelism.

1.2.4 *Hardware generation.*

The last step consists of transforming the nodes of the optimised data flow graph into circuit components during the dynamic programming pass. The technique of dynamic programming is used to generate the alternative hardware configurations. Chapter 4 will cover the dynamic programming while chapter 5 describes the generated hardware.

The generated hardware system appears as decomposed in two interconnected parts: the *control unit* and the *data path* (processing unit). The two units cooperate by exchanging various signals: the *control unit* provides the processing unit with command signals, to inform the latter of the next operation to be carried out. Typically, command lines correspond to control variables of programmable computation resources or to register control. On the other hand the processing unit provides the *control unit* with binary signals called condition variables. These condition variables provide the *control unit* with the relevant information about the past history of the computation to allow decisions about the next step of the computation.

The synthesis can be done using high level primitives such as:

- registers of width n
- adders of width n plus m
- multipliers of width n times m
- n to m multiplexers
- ALU's of width n

That means that no fixed set of hardware modules exists in the library, but there exists a basis set that can be extended according to the specific design needs. Thus for each operator node in the *demand graph* a hardware operator can be generated by a structure generator. This can be done by taking a module from the library, modifying it and combining it with other library modules until the function of the *demand graph* node is attained.

The control synthesis is done during the synthesis of the *data path*. If some operators have to be used twice or more, they have to be multiplexed and controlled. Second, the need for an explicit control of the *data path*, originates from the control nodes. Control synthesis is performed by constructing a finite state machine. Once the *data path* structure is allocated, the control signals are fixed (e.g. load inputs in registers, select inputs in multiplexers, outputs from comparators, etc.). States and state transitions are assigned according to the predecessor successor relation in the *demand graph*. The data path description and the finite state machine description serve as input for the underlying tools in the silicon compiler.

1.3 Related systems

In this section we describe a few research projects, concerning VLSI-design, starting at the highest level of the IC-design: the algorithmic description in a high level language. At Carnegie-Mellon University [Hitch83], [Thom83] and [Black85] research is done on the implementation of behavioral descriptions. Another project is within the Fifth Generation Computer Systems (FGCS) Project in Japan [Mano85]. An expert system is used to translate a description in OCCAM to a CMOS layout. The last research project we will mention is from Carlsruhe University [Camp85], [Rosen85] and [Rosen84].

2. Demand graph construction.

2.1 High level data flow analysis.

For the data flow analysis we want to perform, we can rely on the results of the research done for optimising compilers. The overwhelming majority of previous research in data flow analysis is concerned with low level analysis. Such analysis algorithms act upon a program representation, in which the only control flow structures are conditional jumps [All70]. The structure of the program disappears in the control flow graph representing the algorithm. In a control flow graph nodes represent basic blocks, which are to be executed in linear fashion, and the arcs represent possible flows of control.

But presently new techniques are developed. They operate on a program representation, typically a parse tree or an abstract syntax tree, which includes all of the high level control flow structures present in the source program. High level data flow analysis techniques can be found in [Rose77], [Babi78], [Kenn81] and [Veen85].

The main reason for performing a high level data flow analysis is that the structure of the program is preserved. But there are some other advantages:

- With a good data flow technique it is possible to locate the concurrency of the algorithm represented by the syntax tree. We need this information to be able to exploit the parallelism in the algorithm.

- Several optimisations can be done during the data flow analysis. These optimisations offer the possibility to make the algorithm more suitable for implementation. Very important during the hardware generation is the analysis of dead variables. We must decide which variables have to be stored and which variables are not used anymore at a given moment.

We have chosen the *demand graph* [Veen85] as the representation for our algorithms. The *demand graph method* is used to perform this data flow analysis that results in the *demand graph*.

The *demand graph method* consists of four phases: *syntactic-analysis*, *demand-graph construction*, *application and extraction*. The *syntactic analysis* is performed by the parser, while the demand graph constructor performs the second phase.

The demand graph is a convenient program representation to carry out various flow analysis applications. The *application analysis* consists of depositing initial information in the demand graph nodes and propagating the information through the demand graph, combining the information when appropriate. The analysis has to be concerned only with data flow, since all control flow operators have already been interpreted.

After the demand propagation all information is stored in the nodes and arcs. *Extraction* can take place and all information can be extracted and interpreted in the right manner to be valuable.

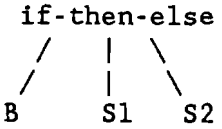
The structure of this chapter is as follows: first general descriptions of the syntax tree and the demand graph are given in the following two sections. Then an example of an algorithm with its syntax tree and demand graph are treated. In the remainder of this chapter the implementation of the *demand graph method* is explained. These sections also contain exact information about the outlooks of the syntax tree and the demand graph in this implementation in CommonLisp.

2.2 Syntax tree.

The *syntactic analysis* is straightforward and converts a program into a syntax tree representation. This analysis is done by a parser. A parser removes all information, that makes the program more readable for humans, but does not contain useful information. The (abstract) syntax tree is a condensed form of the parse tree useful for representing language constructs. The production:

```
S -> if B then S1 else S2
```

might appear in the syntax tree as:



In the syntax tree, operators and keywords do not appear as leaves, but rather are associated with the interior node that would be the parent of those leaves in the parse tree. In this report both the forms parse tree and syntax tree will be used to indicate the abstract syntax tree.

A complete summary of the abstract syntax tree, the demand graph constructor can work upon, is given in Appendix A. An algorithm is a list which starts with the symbol "program". The name of the algorithm is followed by some declarations and the program body. In this body procedures and functions can be declared and called, the usual dyadic and monadic operators can be used and some special control structures can be specified.

Here we will describe some semantics of the syntax tree. These are properties of the language, not reflected in the syntax tree, but determined by the interpretation of the program, made by the *demand graph constructor*.

2.2.1 *Declarations*.

The syntax tree is expected to be free from declarations of variables and constants. These have to be put in special tables when building the syntax tree from the program description. It's expected that the declaration of all variables and constants is checked before building the demand graph. The lists connected to the "program" identifier indicate only which variables are used, so they contain only symbols that are seen as variable or constant names. The symbols that indicate a constant name are identified by the property *value*, which has the value of the constant. This *value* is used in the demand graph instead of its constant name, currently only integer values are supported. Constants may only be declared in the program environment. They can not be declared locally in the procedures.

2.2.2 *Procedures and functions*.

The interpretation of the definition of functions and procedures is made within a global environment. Thus, procedures defined in another procedure may be called from outside that procedure. This is a result of the current implementation but can easily be altered if desired.

2.2.3 *And and Or*.

And and *or* are in essence dyadic operators, but are treated in a special way. When for example the evaluation of the expression **A** in **A** or **B** delivers the true value, expression **B** is not evaluated. Thus we perform a *conditional evaluation* from left to right. The same holds for *and* if the first expression delivers the value false.

2.2.4 *Arrays.*

Arrays are not allowed in the current syntax tree. Considering it is a hardware language, the implementation of arrays has to be one of the first extensions made in the future.

2.2.5 *Types*

There are more constraints on the input language, not determined by the demand graph constructor, but by considering it as a hardware description language. One of these constraints is concerned with the types of the variables. Proposed is to use only one type : integer. You can define the precision of the integer by describing how many bits should be used. This information can be entered in the graph in the *constant* nodes and the *get* nodes. The information can then be propagated through the whole graph, until each processing node knows how many bits it has to process. Thus only at the entrances of the graph (constant and get nodes) you have to specify the bit width. The design system then calculates the bit widths of all the data paths and operators in the data paths. This information is not present in the syntax tree. The parser has to make some additional lists, during the translation of the algorithm to the syntax tree, in which this additional information about the variables is stored.

2.3 **The demand graph.**

The *demand graph* is a graph which describes the data flow in a program. It does not contain any explicit control structures: these have all been interpreted during the data dependency analysis and their effects have been expressed in interface nodes. Interface nodes encode the static ambiguity of data dependency: they appear wherever data dependency is influenced by conditional control flow.

The *demand-graph-construction* transforms the syntax tree in a demand-graph. This is done by adding extra nodes and arcs that encode data dependencies, and by removing control flow nodes that are not essential to the meaning of the program. Nodes that do not in some way construct a new value are not part of the demand-graph: Variable and Assign nodes, for instance, are left out, while a plus node constructs a new value and is therefore part of the demand graph.

2.4 Example: The GCD-machine.

In the example some terms are used that will be declared later. These will become clear when the remainder of this chapter is read. However, the reason the example is presented here is to give the reader an idea of what is going on during the demand graph construction.

The well-known Euclid's algorithm to calculate the greatest common divisor (GCD), is taken as example for the demand graph construction.

The algorithm is described in two input languages Pascal (see fig. 2.1) and LISP. (see fig. 2.2) These descriptions can be translated into the same syntax tree (see fig. 2.3). When we look at the syntax tree, we recognise the function that calculates the remainder. Furthermore, the two while-loops, the get and put operations with their arguments and the call to the function remainder can be found.

This syntax tree is transformed to the demand graph (see fig. 2.4) by the demand graph constructor. In the demand graph we find the data flow of the algorithm. First the two variables a and b are read by the get node. The get nodes represent the IO-protocol needed. These values are entered through entry nodes (EN) in a loop. This loop exchanges the values for a and b and calls the function remainder (call-in nodes) while the output of the test node (NOT) is true. Through the param nodes the values reach the second loop. Here the value of d is unchanged (direct connection between entry and exit node in the rightmost EN-EX nodes). The value of d is subtracted from n each time the loop is traversed, by the - node, as long as the >= nodes output remains true. When false, the value of n is transported through the exit node to the result node and through the call-out node back to the main loop. After finishing this loop, the put node produces the value of a, which is the greatest common divisor of the initial a and b.

```
program gcd (input, output);
var a, b, h : integer;
  function remainder (n, d : integer) : integer;
  begin
    while n >= d do
      n := n - d;
    remainder := n;
  end; {remainder}

begin {gcd}
  readln(a, b);
  while b <> 0 do
    begin
      h := b;
      b := remainder(a, b);
      a := h;
    end; {while}
  writeln(a);
end.
```

PASCAL PROGRAM

Figure 2.1. Euclid's algorithm in PASCAL.

EUCLID'S ALGORITHM

```
(defun gcd (a b)
 (let (h)
  (while (not (= b 0))
   (setq h b)
   (setq b (remainder a b))
   (setq a h)))
  a)

(defun remainder (n d)
  (while (>= n d)
    (setq n (- n d)))
  n)
```

Figure 2.2. Euclid's algorithm in LISP.

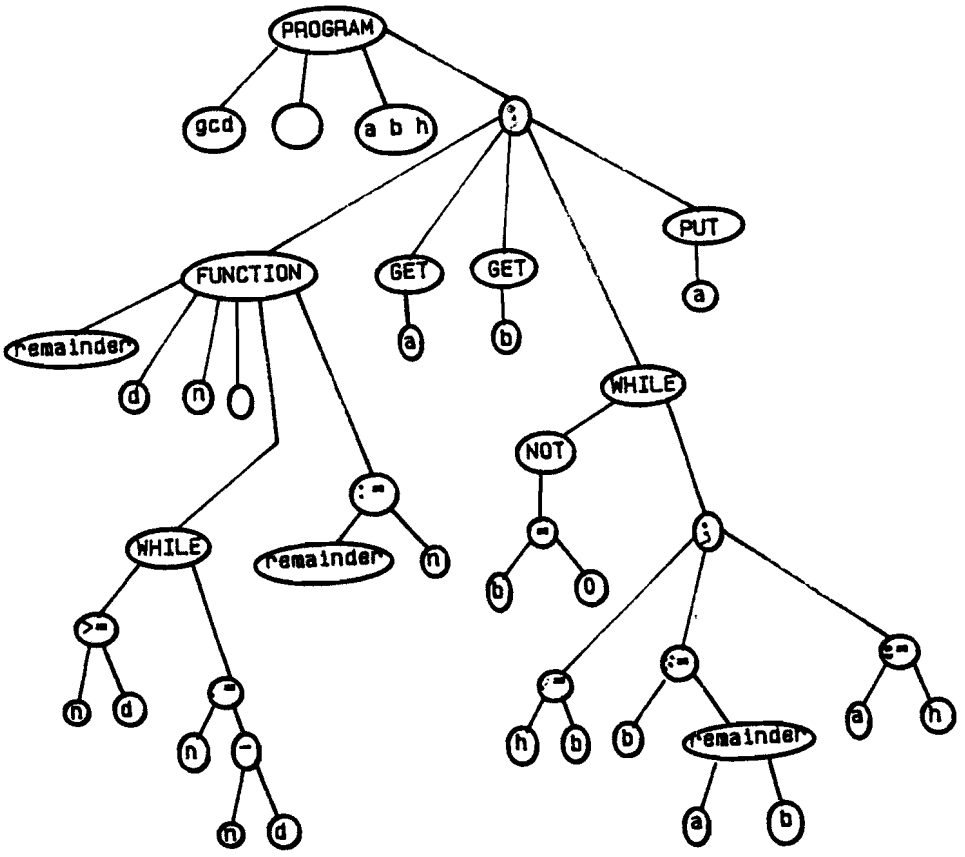


Figure 2.3. Syntax tree for Euclid's algorithm.

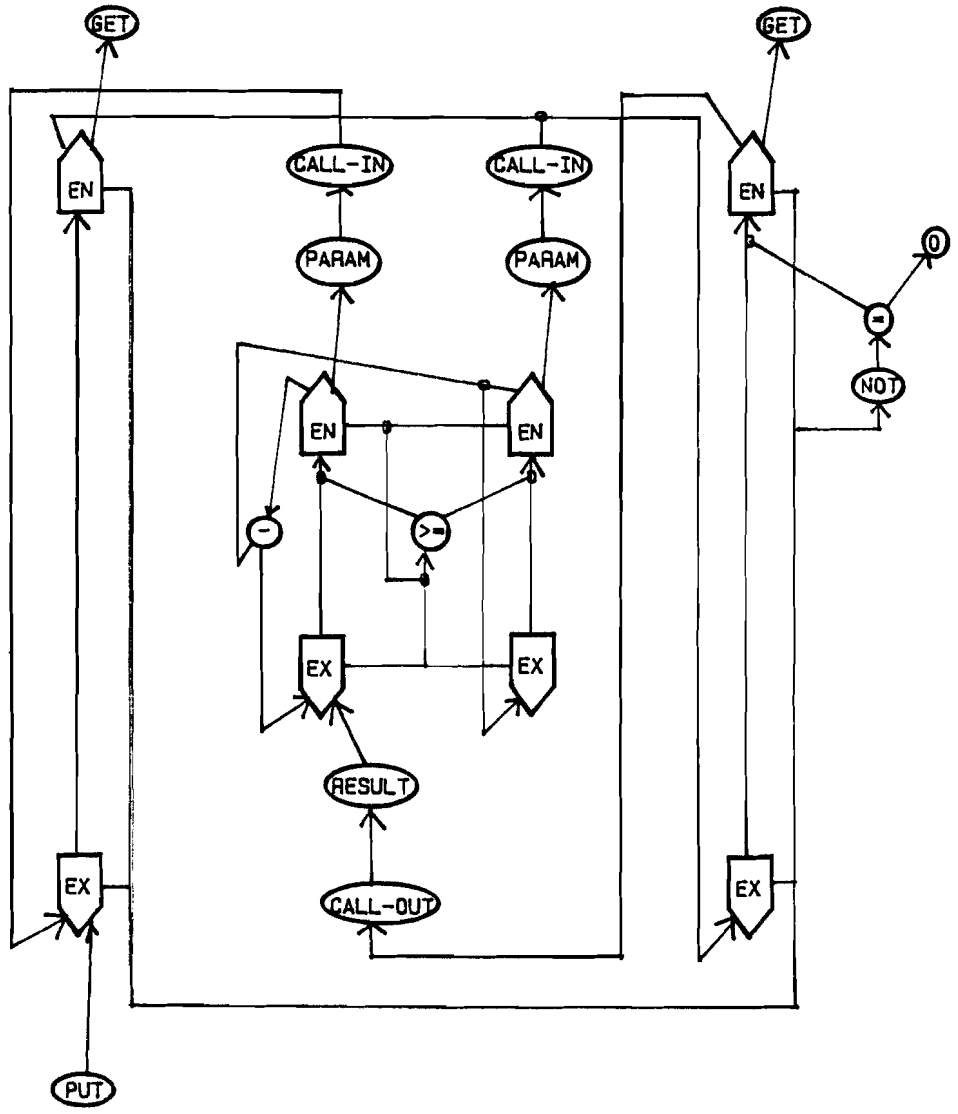


Figure 2.4. Demand graph for Euclid's algorithm.

2.5 Demand graph method.

2.5.1 *Mechanism for descending the syntax-tree.*

The demand-graph constructor has as its input the abstract syntax tree description of the program. The conversion is achieved during a recursive descend of the tree. The algorithm is best understood if each node is considered to be an active object that can alter the graph by adding new nodes and arcs. This process is called attaching the node to the demand graph. The algorithm is implemented by a collection of attach-procedures, one for each kind of node in the syntax tree, including the nodes that will not become part of the demand graph. The construction is started by

attaching the program node and proceeds by recursively attaching all its descendants in an order corresponding to the left to right evaluation order. The descend of the parse tree is achieved by going through the list structure, defining the syntax tree, and calling the appropriate attach procedures.

2.5.2 *Chainer and cocoon mechanism.*

Chainers.

The complicated part of the demand graph construction is the building of the appropriate use-definition graphs. This is controlled by a set of objects called *chainers* and *cocoons*. Each *chainer* contains a *deflist* and an uselist. During the construction one *chainer* is always designated as the current-chainer. In the *deflist* of this current-chainer a variable is stored when it is defined. Defining a variable means: giving the variable a new value. So in the *deflist* are stored the variable name and the node identifier of the node in which it was last defined. In general this will be an assignment node. When a variable is used one can look up in the *deflist* where it was last defined and make a new arc from the use to the definition of the variable. If in a sequential code segment the sequence definition-use-definition-use for one variable occurs, the first use is connected to the first definition and the second use to the second definition. The two definitions are unrelated and the fact that the two groups employ the same variable name has no influence on the demand graph.

In the *deflist* there are other items than variable names. These are called *pseudo-variables*. They are used to store a reference to a certain node. For example, the node identifier corresponding to the pseudo-variable 'Value, is the node which produced the last new value. This is used in assignments where the left hand side has to point to the last produced value of the right hand side.

A variable is said to be *exposed used* if it is used in an environment in which it is not earlier defined. The variable is put in the current-uselist, and an interface node is made for this variable. So the *uselist* contains pairs, with in each pair a variable name and a node identifier. The function of the *uselist* seems a little bit strange at the moment, because normally it is not allowed to use a variable before it is given a value (defined). But in the next section the role of the *uselist* will become clear.

Cocoons.

There are expressions that need special treatment because of their effect on use-definition analysis. For example procedures, loops and conditionals. Whenever during the traversal of the syntax tree such an expression is encountered a new *cocoon* is created. The creation of a new *cocoon* is implemented by making a new *deflist uselist* environment. In this new environment the subgraph corresponding to the expression can be attached in isolation from the remainder of the demand graph. There are different kinds of cocoons corresponding to the different kinds of special expressions. Each special expression contains one or more subexpressions, called branches. For each branch a new *chainer* is created, which is designated as the current *chainer* when that branch is attached. When all branches are analysed a series of separate demand graphs, one for each branch is available. Each branch contains two lists: a *deflist* and a *uselist*. The *deflist* contains the last declaration of all variables within that branch, called the exposed definitions. The *uselist* contains all variables which are used in the branch before they are defined, called the exposed uses.

After all branches have been analysed the *cocoon* is dissolved, which involves the creation of two series of interface nodes, one for the outputs and one for the inputs, and the connection of these to the subgraph and the surrounding graph. For the exposed uses, input nodes are made and these are connected to the use in the branch and to the previous definition in the surrounding graph. For the exposed definitions, output nodes are made and connected to the defining nodes in the branch. They are not yet connected in the surrounding graph but they are put in the *deflist*, corresponding to the surrounding graph, so they can be connected later.

The *chainer* and *cocoon* mechanism for each kind of expression are explained in the next sections where the attaches of all kind of expressions are described.

2.5.3 *Implementations of the attach procedures.*

2.5.3.1 *Implementation in LISP.*

In this section the implementation of the syntax tree and the demand graph in CommonLisp are described.

The syntax tree is implemented as a list in which the arcs are represented by "(", indicating that a new level in the parse tree is entered, and ")", indicating that a level is terminated and the closest higher level is entered again. The exact syntax of each tree element can be found in the

descriptions of the attach procedures.

A special graph structure is developed for the demand graph. A graph is a LISP symbol with two properties: the node-list and the edge-list. The node-list contains the nodes, identified by a LISP symbol with prefix *Node-* and suffix a unique number. The same holds for edges with the prefix *Edge-*. A node has the properties: *type*, indicating the node type (constant, operator), *in-edges*, a list of incoming edges and *out-edges*, a list of outgoing edges. An edge has the *from-node* and *to-node* properties, besides the *type* property.

Furthermore the various stacks and deflists and uselists are implemented as LISP lists.

2.5.3.2 *Form in which the attach-descriptions are given.*

The attach-procedures for each kind of expression that is allowed in the syntax tree are given in the next sections. The descriptions will be presented in two parts:

1. The syntax of the expression in the syntax tree, in EBNF.

2. How the expression is attached to the demand graph.

Some attach-procedures are explained in a figure. The abbreviations used in the nodes have the following meaning:

Dx : Node which defines a variable x.
Ux : Node which uses a variable x.

The following drawing convention is used in the figures: Operator and constant nodes are circled, control nodes look like "houses" and ellipses are special nodes with their meaning in it. (see fig. 2.5).

In the following sections some references to the implementation will be made. Names surrounded by asterisks (*) reference to the names of LISP structures used in the implementation. Names preceded by a quote (') reference to names used in LISP to indicate a certain property or its value.

Detailed information about the demand graph is given in Appendix B.

2.5.3.3 *Attach of constant.*

1. <integer>

2. A node with the value of the constant is created with an outgoing arc, which has the 'type 'source to the sink-node : Node-0 (see fig. 2.5). and the node is placed in the *current-deflist* under the pseudo-variable name 'Value. The sink-node is the node to which all constant nodes are connected. It is only used for initialisation purposes. When a constant node is attached within a special construct, an interface node to the surrounding environment is created. These interface nodes will eventually lead to the sink.

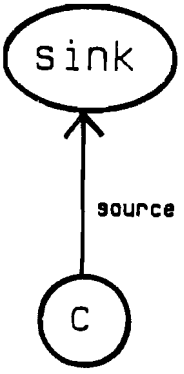


Figure 2.5. Demand graph for constant.

2.5.3.4 *Attach of a symbol.*

1. <symbol>

   If <symbol> is a member of the property list 'constant-list of the *program-name*, it is a name for a symbolic constant and attached as the value of that constant (see previous section). Otherwise it is treated as a variable.

2. If the variable 'is-a-def(inition) then the <symbol> is put in the *current-deflist* with node use('Value), else the pseudo-name 'Value is made to point to the last definition of the variable <symbol> found by use(<symbol>) .

2.5.3.5 *Attach of assignment.*

1. "(" ":=" <left-hand-side> <right-hand-side> ")"

   <left-hand-side> has to be a symbol describing a single variable. <right-hand-side> may be any expression that creates a value which can be assigned to the <left-hand-side>.

2. The <left-hand-side> has to be a single variable because a value is assigned to it. The property 'is-a-def of the variable is set true because the variable is defined here. First the <right-hand-side> is attached and use('Value) contains the node that delivers the value to be assigned to the <left-hand-side>. This is done during the attachment of the <left-hand-side>. See also the next section.

2.5.3.6 *Attach of a sequence.*

1. "(" "\;" {<arg>} ")"

2. Calls the attach procedure for all its arguments from left to right.

2.5.3.7 *Attach of a get.*

1. "(" "get" {<variable>} ")"

   <variable> is a symbol, defined in a variable list.

2. There is a special path for the get and the put nodes. This path represents the succession of the read (get) and write (put) operations specified in the algorithm. This path is controlled by the pseudo variable 'Standard-IO. The first get or put node is connected to 'Node-1 which is the 'IO-sink. The following nodes are connected to their ancestors with a source edge. In this way a path of get and put nodes is formed. When the graph construction is finished the property 'source-of-demands of the *program-name* is set to the last definition of the 'Standard-IO variable. This property marks the beginning of the IO path. The get node is defined in the *current-deflist* together with the variable it gets. Each variable gets its own *get* node. (see fig. 2.6).

2.5.3.8 *Attach of a put.*

1. "(" "put" {<variable>} ")"

   <variable> is a symbol, defined in a variable list.

2. For each variable a put node is made. This node is put in the IO chainer with its left-source edge as described in the previous section. Further the node is connected to the last definition of the variable, that has to be put, with its right-source edge (see fig. 2.6).

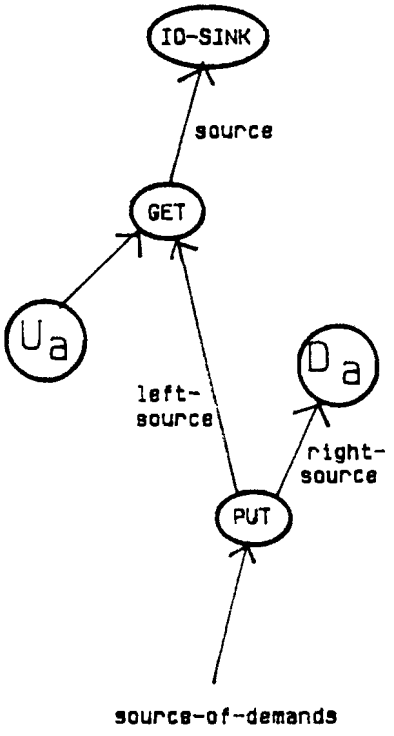


Figure 2.6. Demand graph for put and get nodes.

2.5.3.9 *Attach of a monadic operator.*

1. "(" <monop> <arg> ")"

   <monop> is defined in *monop-set*. <arg> must deliver a value.

2. Makes a new node with 'type <monop> and connects this node to the value delivered by the <arg> with an arc 'source and defines 'Value as the node itself (see

fig. 2.7).

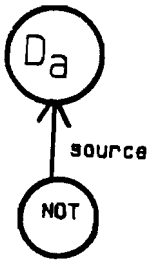


Figure 2.7. Demand graph for the monadic expression: NOT(a).

2.5.3.10 *Attach of a dyadic operator.*

1. "(" <dyop> <arg1> <arg2> ")"

   <dyop> is defined in *dyop-set*. *dyop-set* contains all dyadic operators except for the *and* and *or* operators, treated in the next two sections. <arg1> and <arg2> must deliver a value, acceptable to the <dyop> operator.

2. Makes a new node with 'type <dyop> and connects this node to the value delivered by the <arg1> with an arc 'left-source and to the value delivered by <arg2> with an arc 'right-source and defines 'Value as the node itself (see fig. 2.8).

2.5.3.11 *Attach of an* and.

1. "(" "and" <arg1> <arg2> ")"

   <arg1> and <arg2> must deliver a boolean value.

2. Makes a new node with 'type 'and. This is a branch node. Connects both control and outlink-failure to the value delivered by <arg1> and the outlink-success to the value delivered by <arg2> (see fig. 2.9).

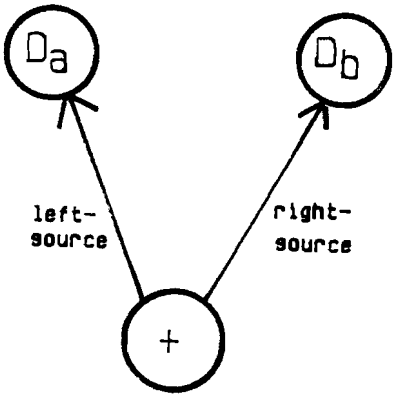


Figure 2.8. Demand graph for the dyadic expression: a+b.

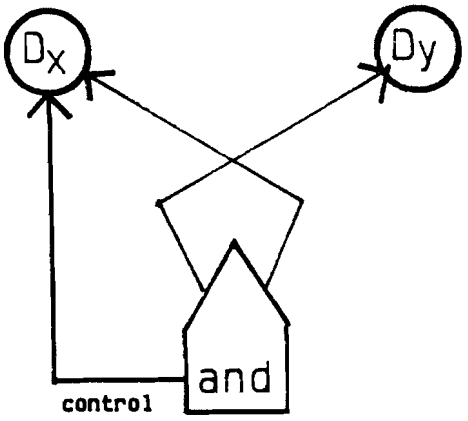


Figure 2.9. Demand graph for the expression: X AND Y.

2.5.3.12 *Attach of an* or.

1. "(" "or" <arg1> <arg2> ")"

   <arg1> and <arg2> must deliver a boolean value.

2. Makes a new node with 'type 'or. This is a branch node. Connects both control and outlink-success to the value delivered by <arg1> and the outlink-failure to the value delivered by <arg2> (see fig. 2.10).

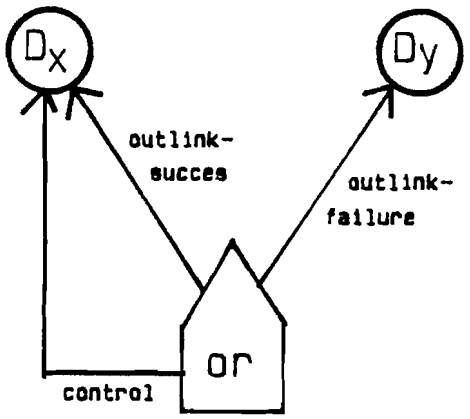


Figure 2.10. Demand graph for the expression: X OR Y.

2.5.3.13 *Attach of conditionals.*

1. "(" "if" <test> <then-chainer> <else-chainer> ")"

   <test> must deliver a boolean value.

   <then-chainer> and <else-chainer> may be nil but may never be omitted. They consist of a statement or a multiple statement.

2. The <test> is attached to the demand-graph in the *current-deflist*. The 'Value it delivers is later connected to the control of the conditional cocoon, but first the conditional cocoon is made. This is done by creating two new deflists and two new uselists. The *current-deflist* and the *current-uselist* are pushed on their stacks. Then the <then-chainer> is attached in the branch-chainer with one deflist and one uselist and the <else-chainer> is attached in the else-branch. For all *exposed-uses* ((see fig. 2.11): U-nodes) new nodes are made. Nodes with 'type 'Link-in-0 for the *then* branch and 'link-in-1 for the *else* branch.

   Now we can dissolve the conditional cocoon. Branch nodes ((see fig. 2.11): B-nodes) are created for all names that occur in some of the two deflists. These branch-nodes are connected to their definition (D-nodes) in the then-chainer with edge 'outlink-success and in the else-chainer with edge 'outlink-failure. If one of the two definitions does not exist then a new node, type 'link-in-0 (definition in *then* non-existent) or 'link-in-1 (definition in *else* non-existent), is made. The 'link-in-1 nodes are put in

the uselist of the *then* branch, and the 'link-in-0 nodes are put in the uselist of the *else* branch. All branch nodes are placed in the *export-list-branch* of the conditional cocoon. Now merge (M) nodes are made for all names that occur in some of the two uselists. The merge nodes are connected with 'inlink-success edges to the 'link-in-1 nodes. If one of these nodes does not exist then it is not connected. All merge nodes are put in the export-list-merge and they are connected to previous definitions (D) in the surrounding demand-graph with 'value edges after popping the deflist and uselist from their stacks. At the end the control is linked to all branch and merge nodes with 'control edges and the branch nodes are defined in the *current-deflist* of the surrounding demand graph.

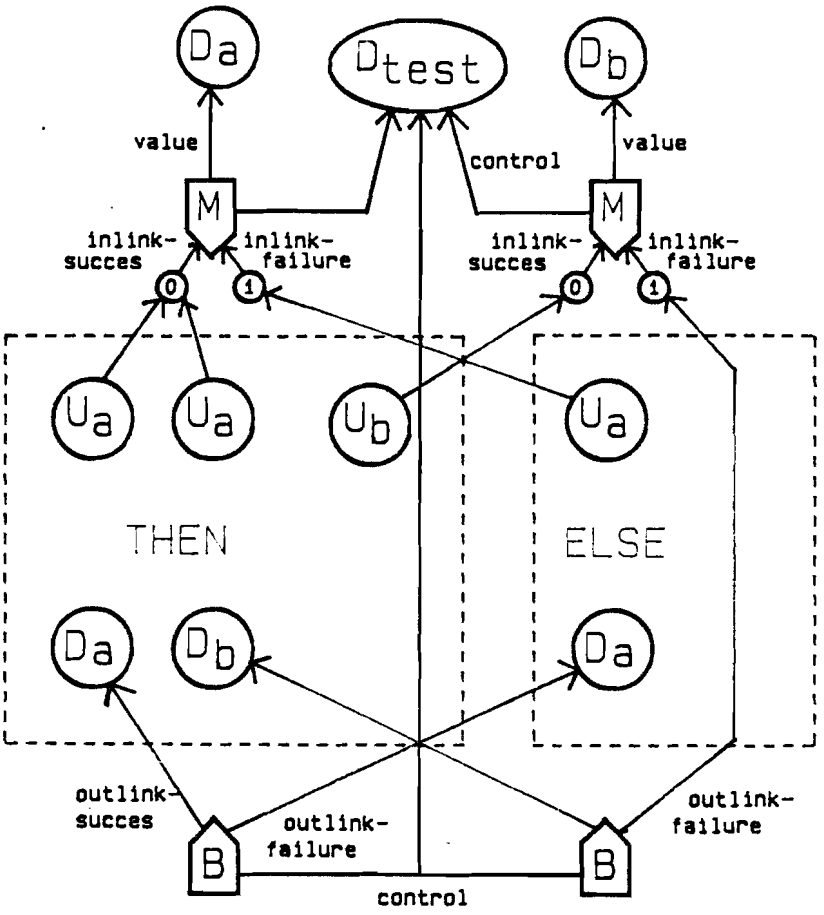


Figure 2.11. Demand graph for if statement.

2.5.3.14 *Attach of loops.*

1. "(" "while" <test> <body> ")"

   <test> has to deliver a boolean value. <body> may be nil but may never be omitted.

2. A loop-cocoon is created. This means that the *current-deflist* and the *current-uselist* are pushed and two new deflists and two new uselists are created. The <test> branch is attached first within its own chainer. The loop-control is set to the value that is created by attaching the <test> branch (see fig. 2.12).

   When there are *exposed uses* in the <test>, nodes of the type 'entry (EN) are made for the concerning variables. Now the <body> branch is attached. If there are exposed uses in the <body> 'link-in-0 nodes are made. All these exposed uses appear of course in the uselist corresponding to the branch in which they are used. When dissolving the loop-cocoon for each name that occurs in some deflist or uselist of the loop-cocoon an 'exit (EX) node is made. The 'exit node is linked to the definition in the <test> with edge 'value. If no definition is available then there will be no 'entry node, thus one is made and put in the uselist of the <test> branch. All names in the uselist of the <test> branch have 'entry nodes. These 'entry nodes are connected to the surrounding graph with an edge 'entry, and with an edge 'last to the definition in the <body>. If there is no definition in the body than a 'link-in-0 node is made, in the same way as if it was an exposed use in the <body>.

   All names in the uselist of the <body> branch have 'link-in-0 nodes. These nodes are connected to the 'exit node with edge 'last. When the loop cocoon is dissolved all 'exit nodes are connected with an 'entry edge to 'link-in-1 nodes. These 'link-in-1 nodes are placed in the *current-deflist* and connections to them can be made, when used later.

2.5.3.15 *Attach of procedures.*

1.

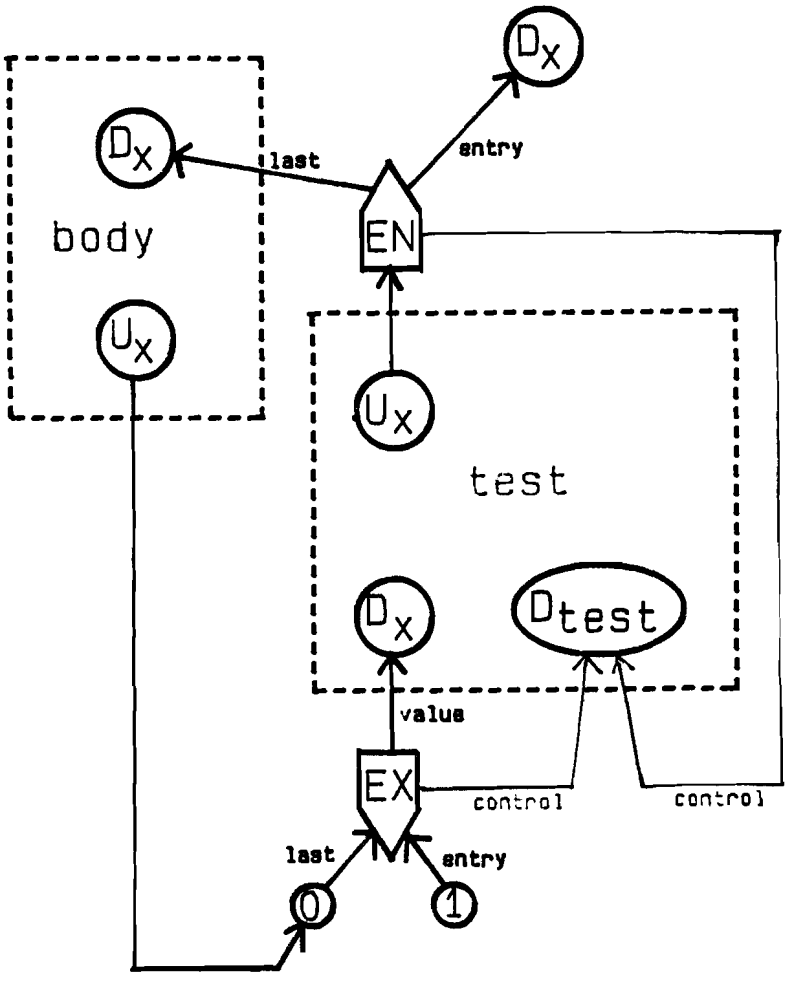


Figure 2.12. Demand graph for while statement.

```
"(" "procedure" <name> "(" <value-params> ")"
                       "(" <reference-params>")"
                       "(" <local-variables> ")"
                       "(" <body> ")" ")"
```

<name> is a symbol, which identifies the procedure.

<value-params> <reference-params> <local-variables> when omitted nil has to be given in their place.

<body> may be nil but may never be omitted. It may be any sequence of expressions.

2. Attaching a procedure starts with checking if there is not already another procedure in the program with the same name. If not, the <name> is stored in the *program-name*'s list 'procedure-list. The sets of <value-params>, <reference-params> and <local-variables> are stored as property lists of the symbol <name>. A new cocoon is created with in it one new deflist and one new uselist. These become current lists when attaching the <body> of the procedure after pushing the other lists. When the body is attached the deflist contains all definitions that have to be exported to the surrounding graph. For all definitions that correspond to reference parameters and global variables 'result nodes are made (see fig. 2.13). These are connected to their definitions with edges of type 'value. Later, when calling the procedure, these nodes can be connected to 'call-in nodes. For local variables and value parameters no result nodes are made because they have no influence on their environment. All 'result nodes are stored in a property list 'outputs belonging to the symbol <name>.

   The uselist contains all exposed uses. Exposed use can occur for <value-params>, <reference-params> and global variables. Exposed uses for <local-variables> are put in a *signal-list* and will be reported when the program finishes. For the other exposed uses, 'param nodes are made while attaching the body. Now these 'param nodes are put in the 'inputs property of the procedure <name>. Dissolving the procedure cocoon is ended with popping the deflist-stack and the uselist-stack.

2.5.3.16 *Attach of procedure calls.*

1. "(" <name> {<param>} ")"

   <name> is a symbol in *programs-name* property list 'procedure-list.

   {<param>} is a sequence with exactly the number of symbols as in the procedure definition are in the <value-params> and <reference-params> lists. The first symbols in <params> are seen as the value params, until no corresponding parameter is found in the <value-params>. The resulting params are reference params.

2. When a procedure is called it is checked if the procedure is already attached. If no error is signaled, for each node in the 'outputs property a

'call-out node is made. This 'call-out node is connected to the 'result node with an edge 'proc-leave. The 'call-out node is stored in the *current-deflist* under the name self if it is a global variable or under the corresponding name in the procedure call heading in case of a reference parameter.

For each node in the 'inputs property a 'call-in node is made. This node is connected to the corresponding 'param node. The 'call-in node is connected to the last definition of the variable in case of a global variable and to the definition of the corresponding name in the <params> list in case of a value param or a reference param (see fig. 2.13).

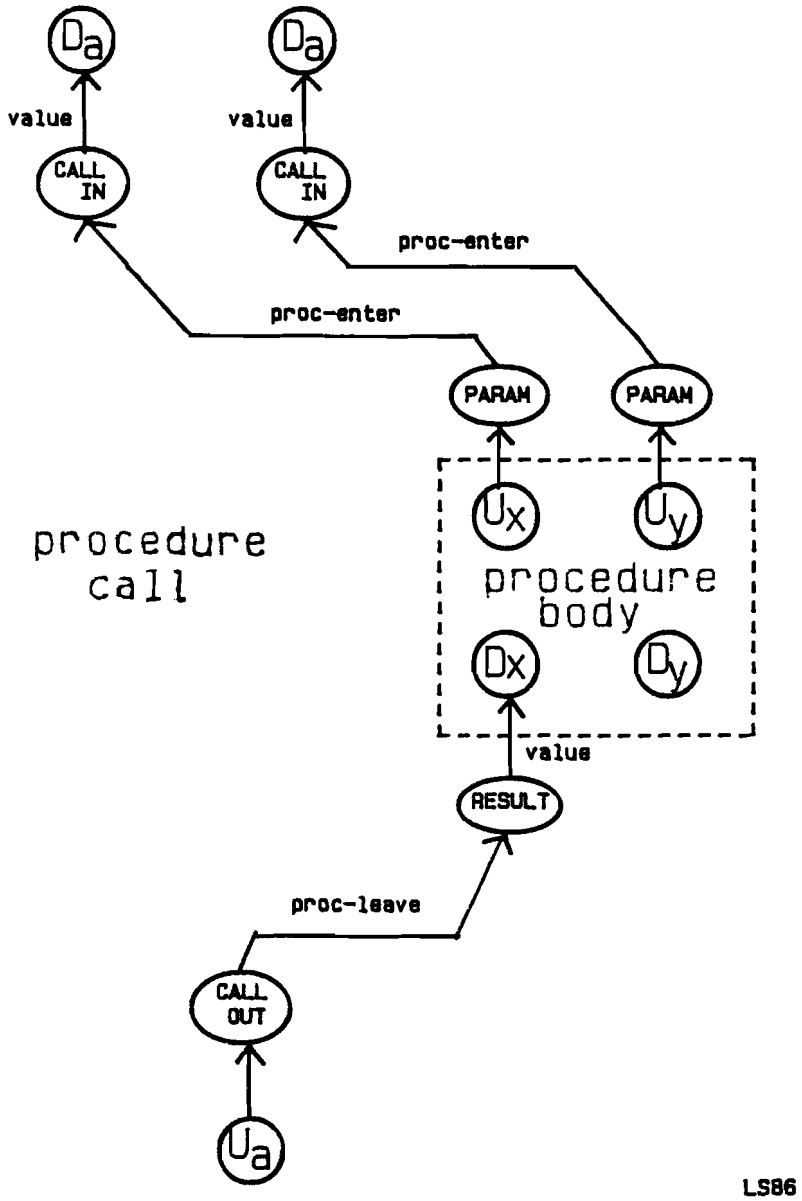


Figure 2.13. Demand graph for procedure with procedure call.

2.5.3.17 *Attach of a function.*

Functions are attached in exactly the same manner as procedures. Only the function name itself is treated as a reference-param when the function exits. Thus a 'result node is made for the variable with the name: function name.

2.5.3.18 *Attach of a program.*

1.

```
"(" "program" <program-name> "(" <constant-list> ")"
                             "(" <variable-list> ")"
                             "( "<body> ")" ")"
```

<program-name> is a symbol that identifies the current program.

<constant-list> ,<variable-list> when omitted nil has to be given in their place.

<body> may be nil but may never be omitted. It may be any sequence of expressions.

2. The <constant-list> and the <variable-list> hold symbols. The lists are stored as properties of the symbol *program-name* namely a 'constant-list and a 'var-list. The identifier *program-name* is set to <program-name>. Then the <body> is attached. The graph that is constructed is stored as a 'graph property of the symbol <program-name>.

3. Applications of the demand graph.

3.1 Introduction

The demand graph can be optimised in many ways. Essential to these improvements is that the demand graph is transformed in an equivalent demand graph. Thus the demand graph represents the same algorithm but its structure is altered, resulting in a better implementation. The outcome is that you can compile the demand graph into hardware at the moment it is generated, or after some improvements. This makes it possible to add more applications on the graph structure afterwards.

3.2 Dead node elimination

Dead nodes are nodes with no predecessors or nodes of whose predecessors are all dead. If these nodes represent dyadic or monadic operators, no following operation requires the values they produce. It is useless to produce these values and the operators can be omitted. If a control node has no more edges that carry values connected to it, it can be removed too. After removing dead nodes it is possible that other nodes become dead and can be removed. Thus when a node is removed, it has to be checked whether any of its predecessors are dead now.

The algorithm : Dead code elimination

1. All nodes without incoming edges are put in a list.

2. From this list one node is taken and it is removed together with its outgoing edges.

3. The list is updated by removing the node processed under 2 and adding new nodes that became dead by removing edges under 2.

4. As long as the list is not empty, goto step 2.

Another approach to eliminate superfluous nodes is the following: *Useless nodes* are all nodes that do not contribute in anyway to the output. In the demand graph the output is represented by put nodes. We check which nodes influence the data, produced by the put nodes, and mark these. All unmarked nodes can be removed afterwards.

The algorithm : Useless node elimination

1. Follow the IO path and for each put node do the Mark-procedure.

The Mark-procedure: For each outgoing edge of a node mark the destination node of the edge and call recursively the Mark-procedure for the destination node.

2. Remove all unmarked nodes.

The second algorithm covers all nodes that would be removed by the first algorithm. The reverse of this assertion is not true.

3.3 Code motion.

A special kind of expressions can be removed from the inside of loops. These are called *invariant-expressions*. An expression is an invariant-expression in a loop if none of the variables in the expression can be modified by execution of the loop. When such an expression is evaluated outside the loop, it is only evaluated once, while inside the loop it may be evaluated many times.

3.4 Remove algebraic identities

Some operations on data do not influence the value of the data. For example the operations:

```
X := X + 0
X := X * 1
X := X / 1
```

can be removed without changing the value of X afterwards. This can be extended to other operators.

3.5 Redundant subexpression elimination.

Repeated operations are the same operations on data, that has not changed meanwhile. Unchanged data in this context means either constants that have the same value or variables that did not change their value. The unchanged variables can be detected quite easily in the demand graph. The outgoing edges of a repeated operator points to the same definition node as the outgoing edge of the first operator. The similarity between constants can be established by comparing their values. When two operators have been classified as being repeated one of them can be removed and its incoming edges can be connected to the other one.

The algorithm : Redundant subexpression elimination.

For each type of operator do:
  Make a list of all nodes with operators of the same type.

For each node in the list do:
If a node is a repetition of one of the other nodes in the
list remove the repeating operator and remove the node from list.

3.6 Constant folding

If all inputs to an instruction are constant whose values are known, the result of the instruction can be computed when traversing the demand graph. The constants are propagated through the instruction. That is why it is sometimes called constant propagation.

Here we shortly list the meaning of the constant folding for the different statements. A full description is given in the section where the graph transformations are covered.

- *Operators*
  The operators and the input constants can be replaced by a new constant with the value that results when the operation is performed on the two constants.

- *If TEST then A else B*
  If TEST of a conditional statement delivers a constant value, one of the branches (A when test is false) is never reached. This branch can be removed from the demand graph.
  There is another possibility for constant folding here. When a variable is defined as the same constant in both A and B it can be moved outside the if statement.

- *While TEST do A*
  There are two possibilities when the TEST of a loop appears to be a constant. First the TEST is false, then the loop is never traversed and can be removed. Second the TEST delivers the true value for ever and a warning can be reported to the designer during the constant propagation.
  Furthermore, when a variable holds the same constant value, during the loop as when entering the loop, it can be defined outside the loop.

- *Procedure (a b)*
  When a and b get the same constant value in all procedure calls the variables can be defined in the procedure.

3.6.1 *Implementation of the constant folding.*

The algorithm for constant folding is described below:

The Algorithm : Constant folding

```
PILE := all constant nodes in the demand graph;
while PILE is not empty
  Take a NODE1 from the pile;
  Remove NODE1 from the pile;
  For each input-edge of NODE1
    NODE2 := start node of the input-edge;
    If PROPAGATE-THROUGH(NODE2)
       put NODE2 on the PILE;
  REPLACE(NODE1);
```

The procedure PROPAGATE-THROUGH(node) delivers the value true if the constants can be propagated through the node. The criteria for this propagation are given in the following section. The procedure REPLACE(node) replaces the node by the above described structure and performs the actions.

The function "const-propagation" returns a list of all the nodes through which constants are propagated. As a side effect it alters the demand-graph by removing these nodes and replacing them by equivalent structures.

There is another algorithm for finding the nodes through which constants can be propagated. The only entrance for variables in the graph are the get nodes. Thus, when we start a mark procedure, similar to the mark procedure of the dead code elimination, from the get nodes we can find all nodes that can be reached from the get nodes. The remaining nodes cannot be reached from the get nodes and cannot be supplied with variables. Through these nodes constants can be propagated.

The algorithm : Find foldable nodes

1. Follow the IO path and for each get node do the Mark-procedure.

   The Mark-procedure: For each incoming edge of a node mark the departure node of the edge and call recursively the Mark-procedure for the departure node.

2. Through all unmarked nodes constants can be propagated.

3.6.2 *Graph transformations during constant folding.*

Constant propagation asks for special actions for each node type. A special description has been developed to describe these actions. First this description will be introduced.

- Node names are surrounded by *'s, for example *dyop* means a node which represents a dyadic operator.
- In front of the node name its output edges are given in a list surrounded by "(" and ")".
- Behind the node name the input edges are given in a list similar to the output edges.
- Edges inside "[" and "]" mean that the node to which this edge leads has to be a constant node.
- L --> R means: L is transformed to R
- The actions which have to be done during the transition --> are described in between "{" and "}".

We will describe the transformation of a DYOP node as an illustration to the transformations given below.

When a DYOP its outputs, left-source and right-source, both lead to a constant node (indicated by the brackets "[" and "]") then this node can be replaced by a constant node C. The inputs of the DYOP node (V1..Vn) are connected to the constant node C. The calculation of the constant C is done by applying the function of DYOP to both constants, as indicated by the action inside the brackets "{" and "}".

Here follows the table with the descriptions for each node type.

DYOP:
([left-source] [right-source]) *dyop* (V1 .. Vn)

--> *C* (V1 .. Vn)

{ *C* = function of *dyop* (left-source,right-source) }

MONOP:
([source]) *monop* (V1 .. Vn) --> *C* (V1 .. Vn)

{*C* = function of *monop* (source) }

AND, *OR*:
(outlink-success, outlink-failure, [control]) *and/or* (V1 .. Vn)

--> *N* (V1 .. Vn)

{ if control=1 then *N* = ('to-node of outlink-success)
else *N* = ('to-node of outlink-failure) }

BRANCH(1):
(outlink-success, outlink-failure, [control]) *branch* (V1 .. Vn)

--> *N* (V1 .. Vn)

{ if control=1 then *N* = ('to-node of outlink-success)
else *N* = ('to-node of outlink-failure) }

BRANCH(2):
([outlink-success], [outlink-failure], control) *branch* (V1 .. Vn)

--> *C* (V1 .. Vn) | ϵ

{if VAL ('to-node outlink-success) =
VAL ('to-node outlink-failure)
then *C* = 'to-node of outlink-success
else nothing happens }

MERGE(1):
(value, [control]) *merge* (inlink-success, inlink-failure)

--> *C* (inlink-success) | *C* (inlink-failure)

{ *C* = 'to-node of value;
If control=1

```
    then  *C* (inlink-success)
    else  *C* (inlink-failure);
   delete (other-link) }
```

Remark:
Delete means removing all nodes, starting with 'from-node of delete-link until a branch node with the same control as the merge-node is reached.

MERGE(2):

```
([value], control) *merge* (inlink-success, inlink-failure)

                --> *C* (inlink-success, inlink-failure)

    { *C* = 'to-node value }
```

LINK-IN-0 or LINK-IN-1 or CALL-IN or CALL-OUT:

```
([value-in]) *node* (value-out) --> value-out = value-in

                { remove link-in-node }
```

Remark:
node = link-in-0 or link-in-1 or call-in or call-out

ENTRY(1):

```
(entry, last, [control]) *entry* (value)

                      --> (error | entry-exit = entry)

 {if control=1 then error: endless loop
              else (entry-edge of exit)=entry, delete-loop}
```

ENTRY(2):

```
([entry], last, control) *entry* (value) --> *C* (value)  | ε

  {if type('to-node last)= type('to-node entry) or
                         = link-in-0
    then *C* = 'to-node entry
    else nothing happens}
```

Remark:
Delete-loop deletes all nodes in between an entry and an exit loop.

EXIT:

```
(value, [control]) *exit* (last, entry)

                    --> (value, [control]) *exit* (last, entry)
```

PARAM:
([proc-enter1]..[proc-entern]) *param* (value) --> *C* (value) | ϵ

```
{ if     [proc-enter1]..[proc-entern] point to nodes with the
         same value
   then  *C* = 'to-node of proc-enter1;
         delete ('to-nodes of proc-enter2..proc-entern);
   else  nothing happens }
```

RESULT:
([source]) *result* (V1 .. Vn) --> *C* (V1 .. Vn)

{*C* = 'to-node of source) }

4. The dynamic programming approach

4.1 Introduction

In general dynamic programming is used to generate a limited number of solutions to a problem. But in this set of solutions the optimal solution has to be present. To be sure the optimal solution is in the set, we need a quantity in which the optimum can be expressed. This quantity is called the return. Suppose we have available a certain quantity of a resource. This abstract term may represent the area on an Integrated Circuit. A conflict of interests arises from the fact that a resource can be used in a number of different ways. Each such possible application is called an activity.

As a result of using all or part of this resource in any single activity, a certain return is derived. The return may be expressed in terms of the resource itself, or it may be measured in entirely different units. The magnitude of the return depends both upon the magnitude of the resource allocated and the particular activity.

The basic assumptions are:

1. The returns from the different activities can be measured in a common unit.

2. The total return can be obtained as the sum of the individual returns.

The fundamental problem is that of dividing our resources so as to maximise the total return.

It is impossible to investigate all possible implementations of a demand graph in hardware and realise the optimal one. But we still want to obtain an optimum. How is this possible?

The problem as defined above is a *multistage decision process*: a process in which a sequence of decisions is made, the choices available being dependent on the current state of the system-that is, on the previous decisions. For the hardware generation problem, the decision at each stage is which node to implement next. In such processes the problem is to determine the optimal sequence of decisions- that is, those that minimise (or perhaps maximise) some objective function. In the solution of such problems by *dynamic programming*, we rely on the principle of optimality:

Principle of optimality:
An optimal policy has the property that whatever the initial

state and the initial decision are, the remaining decisions must constitute an optimal policy with regard to the state resulting from the first decision.

By a *policy* is meant a sequence of decisions.
Applying this principle to the solution of combinatorial problems essential means using the decomposition principle: the solutions to subproblems are found and then used to find solutions to larger subproblems and, finally, to the problem itself.

An exhaustive description of dynamic programming can be found in [Bell57] and in [Bell62].

4.2 Generation of states.

First we describe what exactly a *state* is. A state is characterised by a set of demand graph nodes. How this set is formed is described in the following section. For the moment it is enough to know that node can be added to and deleted from a set belonging to a state, thus generating new states .

For example: (see fig. 4.1)
In State-0 there are three nodes in the set, consequently three new states S-1, S-2 and S-3 are generated. Now the sets of nodes are calculated for the states S-1, S-2 and S-3 and the process continues.

Secondly to each state a *cost* is added. How these cost are calculated is treated later. The *cost* of a state is used to eliminate the generation of equal subtrees in the dynamic process. When deleting and adding node from and to a node set of a state delivers a new states, it is possible that this new state is generated by operations on the set from another state.

For example: (see fig. 4.1)
Suppose S-4 is characterised by the implementation of the nodes Node-1 and Node-2. When in State-0, Node-1 is implemented during the transition to state S-1 and Node-2 during the transition to state S-2, state S-4 is reached from S-1 (implementation of Node-2) and S-2 (implementation of Node-1).

The cost is used to choose the best preceding state for the new state, this is the state with the lowest cost.

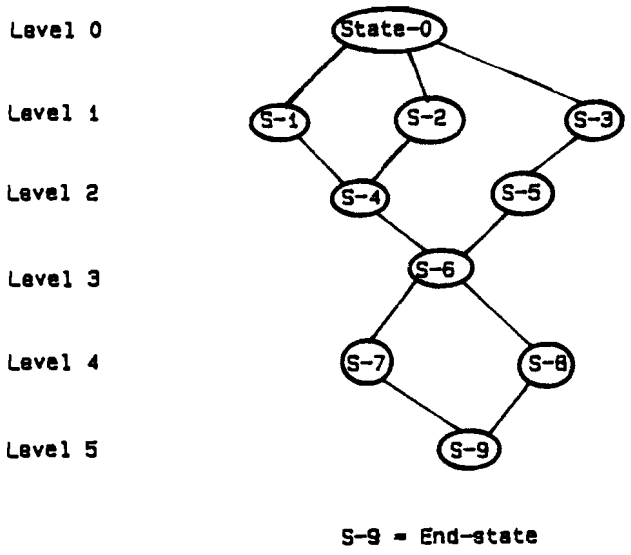


Figure 4.1. Dynamic process lattice.

In this chapter is described how the lattice is build. We assume that an *implementation* can be made for each node and that the *costs* are returned. The *implementation* of a node means that hardware is generated for it. How the implementation for the various nodes is made and the costs are calculated described in the following chapter. Here we assume that functions are available to implemented each requested node and return the costs of that implementation.

4.3 Model definition.

In this section a model of the problem of hardware generation is presented that makes it possible to use a dynamic programming approach to extract hardware.

4.3.1 *Allowed decisions*.

The *allowed decisions* determine the number of states that are generated. There are two contradictory constraints:

- Enough states must be generated to make sure that the optimal end state is reached.
- As few states as possible must be generated to delimit the time in which the implementation can take place.

Before we proceed to the description of the model we introduce a few definitions.

Free:
A node is free if all the predecessors, that produce values the node needs for his operation, are implemented.

We call the set of inputs that a node needs for his operation: the *inputs needed for operation*. For short this is indicated as the *needed set*. It is clear that the *needed set* is different for each node type. The different needed sets are described below.

Implementable:
A node is implementable if it is free and if all nodes that are of the same type and belong to the same control structure are free too.

The nodes of the same type as the main node and belonging to the same control structure are called the *related node set* of the main node.

As indicated before a state is characterised by the set of nodes it contains. We call this set the *bucket*. The *bucket* contains the nodes that are *free* in the state. With the previous definitions we can define the *allowed decisions*.

Allowed decisions:
Given a state and a bucket belonging to this state. A new state may be generated for each node (or set of related nodes) in the bucket, that is (are) implementable.

We complete this section of the model definition by given the *needed set* and the *related node set* for each node type. If the first (second) set is empty the node is free (implementable).

1. **Sink and IO sink**
   Both sets are empty thus may always be implemented.

2. **Operator nodes**
   Needed set: nodes at all outgoing edges.
   Related node set: empty.

3. **Merge**
   Needed set: nodes at control line and outgoing edges.
   Related node set: other merge nodes with the same control line.

4. **Branch**
   Needed set: nodes at control line and both outlinks.
   Related node set: other branch nodes with the same control line.

5. **Entry**
   The entry nodes appear two times in the bucket, first when the loop is entered (entry-1) and second when we traverse the loop itself (entry-2).
   Entry-1
   Needed set: nodes at the outgoing edge from outside the loop.
   Related node set: other entry nodes with the same control line.
   Entry-2
   Needed set: nodes at the outgoing edge from inside the loop and the control line.
   Related node set: other entry nodes with the same control line.

6. **Exit**
   Needed set: nodes at the outgoing edge and the control line.
   Related node set: other exit nodes with the same control line.

7. **Call-in**
   Needed set: node at one outgoing edge.
   Related node set: other call in nodes connected to the same procedure call node, with an implemented node on the, to the needed set node related, outgoing edge.

8. **Result**
   Needed set: node at the outgoing edge.
   Related node set: other result nodes belonging to the same procedure.

4.3.2 *Cost functions.*

This part of the module definition is related to the hardware generation and therefore treated in the following chapter. Here we assume that functions exist to implement a node in hardware. These functions return the costs for this implementation. The cost of a state is the sum of the cost of the preceding state and the cost of the implementation of the node, that is implemented during the transition. When two states deliver the same new state the cheapest path leading to this new state is saved. The other one is removed. Consequently, by eliminating a subtree we delimit the number of states and proceed only with these states that provide a sub optimum.

4.4 The algorithm for generation of states.

In this section the algorithm to generate the set of states for a demand graph is presented.

The Algorithm: State generation.

```
Initialise;
repeat
  current-state-list := new-state-list
  new-state-list := nil
  repeat
    Process-state (car current-state-list)
    current-state-list := (cdr current-state-list)
  until current-state-list is empty
until new-state-list is empty
```

The Initialise procedure:
We start the dynamic process by making an initial state (State-0) with the first "get" node and the constant nodes, as its bucket. State-0 is entered in the new-state-list.

The Main routine:
The outer loop is entered and meanwhile the current-state-list is defined and the new-state-list is emptied. The inner loop is used to perform all operations once for each node in the current-state-list. We take one node from the current-state-list, it is called the current-state. This current-state is processed in the Process-state procedure. When entering a new iteration of the outer loop the current-state-list is set to the new-state-list. This causes the horizontal levels in the lattice (see fig. 4.1). Each time the current-state-list is set to the new-state-list a new level in the lattice is entered.

The Process-state procedure:
Process state performs several tasks: it generates new states and meanwhile it implements the nodes in hardware. The bucket belonging to this state is called the current-bucket. The following is done for each node in this bucket.

> It is checked if this node is implementable. If so all alternative implementations are generated for this current-node and the optimal implementation is chosen. All its successor nodes, freed by this implementation, are put in a new-bucket together with the nodes in the old-bucket, except for the current-node. A new state in the dynamic process is formed. The new-bucket is stored as the bucket of the new formed state. The

states get a successive number. The new generated state is stored in the new-state-list.

When this process finishes there are implementations for all nodes that were in the current-bucket. Thus we have generated as many new states as there were implementable nodes in the current bucket. This is only true if no new states would be the same. Then less states are formed. If a new state has to be generated it is first checked if this state is already existent.

4.4.1 *Algorithm efficiency*

It is clear that the number of states depends on the number of nodes in the demand graph. The purpose of the dynamic programming technique is that all possible optimal hardware structures are examined. If the algorithm is highly parallel many states will be generated. This is inherent to the programming strategy. Our aim was to delimit the number of states that are generated thanks to the special nodes that are added during the demand graph construction. The special nodes generate a few states more in the length of the process but the process does not grow wider. This is important because the width of the process determines the number of states that have to be stored at one time and indicate that many new states can be generated.

There are two mechanisms that delimit the number of states that exist at on level. First, we have the detection of the same states that eliminate sub lattices by determination of a sub optimum. Given the definition of the bucket, the same states can only be generated at the same horizontal level (see fig. 4.1) in the process. This is because the same nodes have to have been implemented for states to be the same. They are only implemented in a different order and that is why they could have generated other hardware. Consequently we only have to check the states of one level to determine if a state is already existent. This mechanism also provides one end state in which all the nodes of the demand graph are implemented.

The other mechanism is provided by the special restrictions, used when it is determined if a node is implementable. They synchronise the dynamic process at certain points and greatly delimit the number of states. For example when there are call-in nodes in a bucket they are not implemented before all call-in nodes belonging to one call are in that bucket.

4.4.2 *Implementation of the dynamic programming*

In Appendix C is described how each node type is treated during this process. This description is based on the implementation of the demand graph as described in the previous chapters. It can be used by readers who have to deal with the current implementation of the algorithm, or serve as an illustration of the principles presented in this chapter.

4.5 **Example**

We continue the GCD example of the preceding chapter. We will demonstrate the dynamic process using the demand graph for the GCD algorithm (see fig. 4.2).

In figure 4.1 is the lattice of the dynamic process for the GCD algorithm given. We will give the explanation with references to these two figures.

The bucket of State-0 is formed with the nodes that are free considering Node-0 (sink) and Node-1 (IO-sink) are already implemented. These nodes are: Node-19 (freed by the *sink*) and Node-14, the initial IO-node, freed by the *IO-sink*. The only node from this bucket that can be implemented is Node-14. This implementation frees the nodes 27 and 15. In State-2 both *get* nodes are implemented and the *entry* nodes, belonging to the same loop, are free. All three nodes are implemented at once and Node-18 is the only freed node. When, in State-4, the loop control node, Node-17, is done, all exit nodes are free. The new bucket formed contains the *entry* nodes 27 and 19, and the two *call-in* nodes 21 and 23.

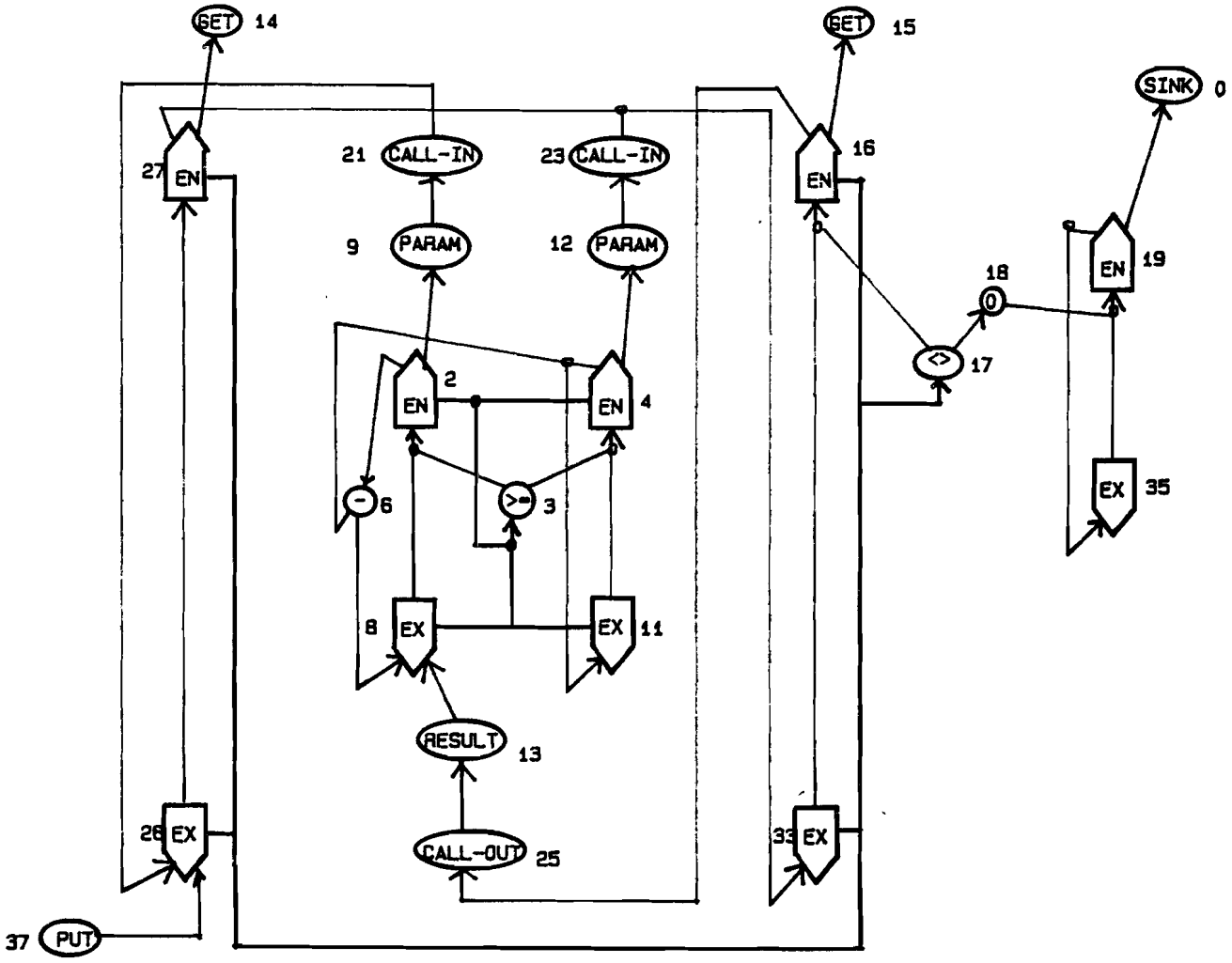


Figure 4.2. Numbered GCD demand-graph.

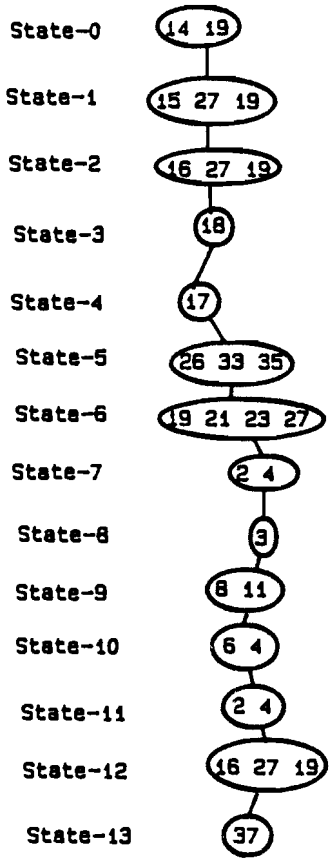


Figure 4.3. Process lattice for the GCD demand-graph.

We push nodes 27 and 19 on the procedure-stack and start implementing the procedure. When we implement both *call-in* nodes we can pass by the param nodes and the freed nodes are node 2 and 4 in State-7. This loop is implemented in a similar way as the main loop described above, resulting in a bucket that contains only the result node 13 in State-12. Implementing the *result* node, passing by the *call-out* node and popping the procedure-stack results in the bucket of state 13. Now we have all three *entry* nodes for the second time and we can close the main loop. This is done by restoring the free node after the exit nodes: Node-37, and the process ends with the implementation of this node.

As we can see in the foregoing example an algorithm without parallelism delivers only a straight line lattice. The additional nodes in the demand-graph, to represent the control flow, do not generate a wider lattice. This is a very important result illustrating the complexity of the dynamic process and the storage capacity needed.

This example was meant to give an idea of what is going on during a pass through the demand graph.

5. Hardware synthesis.

5.1 Introduction.

In this chapter the hardware generation is treated. In each state, described in the previous chapter, hardware must be generated for a single node or a collection of nodes. The demand graph nodes are implemented with modules that are provided by the module library. At the same time this library provides the costs for the implementation. When a node has to be implemented a few alternatives are tried. For each alternative the additional hardware is calculated and the costs are given to a selector. This selector chooses the cheapest implementation and the hardware structure of the state is expanded with the new hardware. If necessary the state machine is adapted for the new hardware at the same time.

We will first outline a few difficulties that arise during the hardware generation. Further the outlooks of the hardware and the state machine are described. The cost calculations are treated next. The remainder of this chapter covers the hardware transformations for the demand graph nodes.

5.2 Difficulties during hardware generation.

The synthesis algorithms transform the algorithm to real circuits. Problems arise from the difference between the constraints on the algorithm at one side and the constraints on the hardware at the other side. We will outline a few difficulties that arise from this controversy here. Some difficulties are solved during the hardware generation. More technology dependent difficulties have to be solved at a lower level in the silicon compiler.

1. In counterpart to the specification of the algorithm the signals can not be used on as much places as you want in the real circuit. Limits are dependent on the technology used and the timing characteristics given by the designer.

2. Outputs of different components can not be connected in each manner. When no special precautions are taken, the creation of a short-circuit can lead to unpredictable results.

3. Unwanted feedback can be created when in the specification outputs are used, that are inputs in foregoing operands, with only combinatorial logic in between them. Special synchronisation has to take

place in this case.

4. Both a point in the time space and a place in the area space must be found to implement an operation in hardware. This transformation in time and space has a special meaning when an operator is used twice or more. Each time an operator is implemented, one has to weight multiplexing an operator against placement of a new operator.

5.3 The processing unit.

First we have to define a description of the hardware. The hardware is split in two parts: the processing unit and the control unit. In this section the processing unit will be described. In the next section the control unit will be covered.

In each state of the dynamic process the hardware must be described. Thus the description must be as short as possible without the loss of vital information for the coming hardware generation.

We describe the hardware in two lists. The first list contains all hardware nodes and to which nets their ports are connected. The second contains a net list,with for each net all nodes, this net is connected to. The two lists together form the hardware list (hdw-list). Two lists are used to delimit the number of search operations, that would have been done when one list was used.

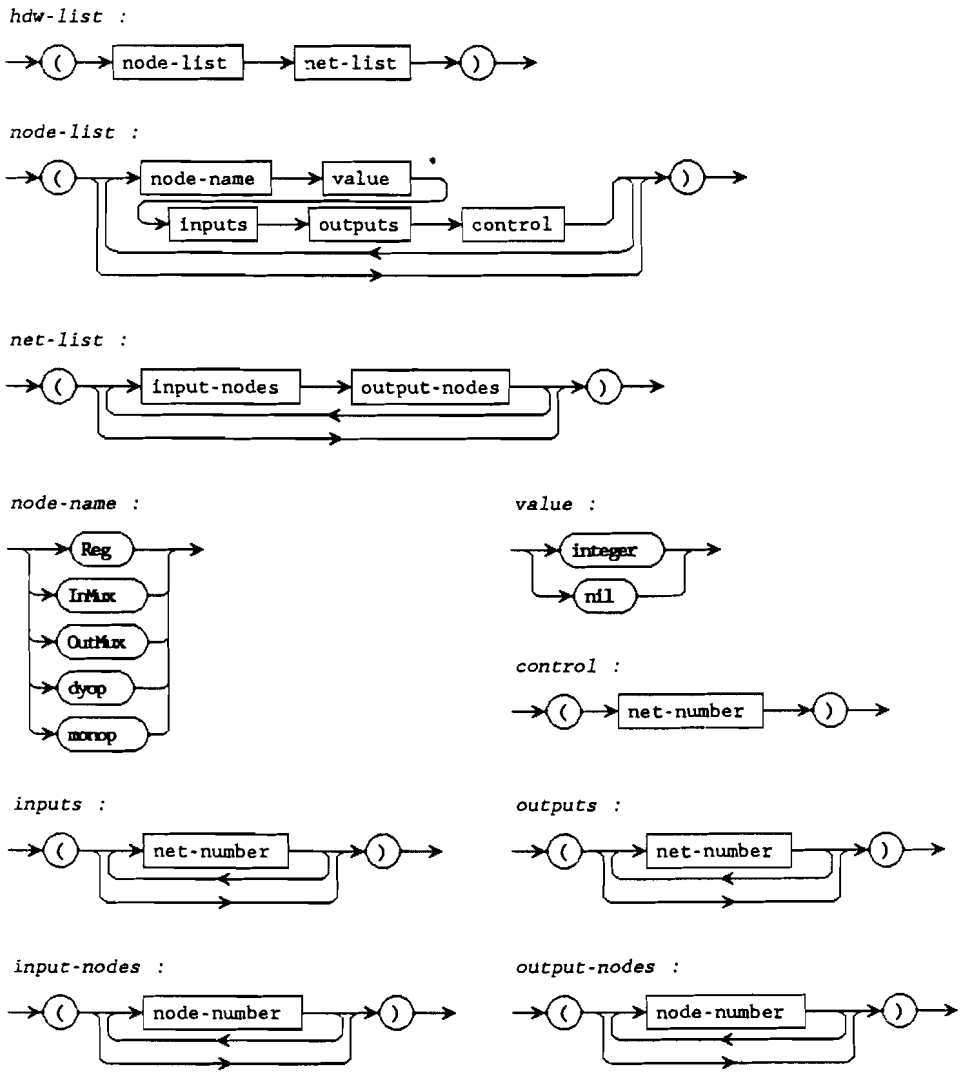


Figure 5.1. Hardware structure.

The various hardware nodes are registers, multiplexers and operators. Each type of node has its own number of in- and outputs. They are described in the following list.

```
Register          input-nets : 0    : register-input
                  output-nets: 0    : register-output
                  control-net: 0    : clock-in signal

IN-multiplexer    input-nets : 1..n: inputs
                  output-nets: 0    : output
                  control-net: 0    : input selector

OUT-multiplexer   input-nets : 1    : input
                  output-nets: 0..n: outputs
                  control-net:     : output selector

operators         input-nets : 0..k: inputs
                  output-nets: 1    : output

multi-operators   input-nets : 1..k: inputs
                  output-nets: 1    : output
                  control-net: 0    : function selector


                  n: dependent on the type multiplexer.
                  k: dependent on the type operator.
```

5.4 The control unit.

The control unit is a finite state machine. The state machine is represented by a LISP list. The state machine is called a cycle-list because each label represents a new cycle in the state machine. We omit the term "state" for each state in the state machine and use the term "cycle" instead, to prevent confusion with the states of the dynamic process. The construction is as follows:

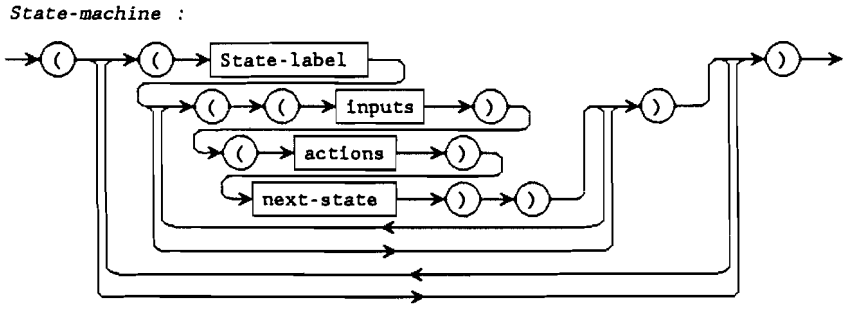


Figure 5.2. The control unit

Each cycle has a cycle label. For each cycle that can be a successor of these cycle a list is made, containing the inputs and the actions belonging to this cycle transition and the new cycle label. In this way we have created a Mealy machine in which the old cycle and the inputs both determine the new cycle. The inputs and actions are pairs of a net number and a value. The value is the actual value that must be put on the net during this cycle transition.

In this way we have created a state machine in which we can express the cycle transitions that have to take place due to the special language constructs. For example:

the construct IF x THEN a ELSE b

could be translated into

```
(<S1> ((x . 1) (actions a) S2)
      ((x . 0) (actions b) S3)
 <S2> ...
 <S3> ...
 )
```

and the construct WHILE x DO a

could be translated into

```
(<S1> ((x . 1) (actions a) S1)
      ((x . 0) (actions) S2)
 <S2> ...
 )
```

5.5 Hardware description and register transfer languages.

When we look at the hardware descriptions given above, we see a close relation to register transfer languages. It is easy to combine the description of the state machine with that of the process unit and automatically generate a description in some register transfer language. For example: DDL (A Digital System Design Language) [Duley68]. From this description we can use some other tools to synthesise the final system. For example Takagi at NTT has build a system that translates DDL descriptions into hardware [Taka84]. He uses DDL-S, a LISP based DDL. The -S stands for the LISP's S-expression syntax he uses. This syntax is extremely simple to interface with our hardware description.

This illustrates one of the interfaces we can use to lower levels of the silicon compiler. The expression of the generated results in a register transfer language has the advantage that we can develop parts of our design using the high level system and interface this with other parts of the design done in the register transfer language.

5.6 Cost calculations.

5.6.1 *General cost functions.*

Here we give a presentation of what a cost function could be. No research has been done on the cost functions yet. We merely present them in this section to give the reader an idea of how cost functions could be defined. Each cell in

the module library has costs for its size, dissipation and speed.

The cost function R(S) we have to optimise is:

$$R(S) = C_1 * \sum_{e \epsilon S} T_e + C_2 * \sum_{e \epsilon S} D_e \qquad (5.1)$$

with C_1 , C_2 : weight factors.
D_e : Dissipation by hardware element e.
T_e : Time delay caused by hardware element e.
S_e: Set of hardware elements in this implementation.

under the constraint:

$$\sum_{e \epsilon S} A_e \leq \text{Total area}$$

with A_e : Area needed by hardware element e.

We can choose the area of interest we want to minimise most by choosing the right values for C_1 and C_2. For example: when we set C_1 to zero we can optimise to dissipation alone.

Now we have to define a function that calculates the costs for each state during the process: We define the costs in state-n f_{stn} as follows:

$$f_{stn} = \min_{\{s \epsilon s_{pre}\}} [f_s + C_1 * T_{e_s} + C_2 * D_{e_s}] \qquad (5.2)$$

with:

f_s : the cost in state-s.
s_{pre}: the set of all states preceding state-n that can evolve to state s_n when one element e_s is implemented.
e_s : element that is implemented.

It is obvious that:

$$f_{st0} = 0 \qquad (5.3)$$

5.6.2 *Implementation costs.*

Up till now the cost functions are defined during the state transitions. But we have to choose between alternative implementations when a node is implemented.

When we have to add a new hardware element we calculate the cost for each possible implementation. When an operator is added we can generate a list like the following:

OPERATOR_NAME

implementation:	cost
processor_1	K1
processor_2	K2
processor_3	K3
...	
processor_n	Kn

This list represents the different implementations for the OPERATOR_NAME. It can be implemented in n-ways by n-different processors. Similar lists can be made for registers when a value has to be stored. From this list we can take the optimal implementation. We get this list with alternative implementations by checking the following possibilities:

1. A new processor, performing the function of the operator, can be added.

2. An existent processor with the same function can be used. The cost for additional multiplexers and so on, are taken into account in the cost.

3. An existent processor can be altered to perform both the old functions and the new function. Then the cost for the altered processor and additional multiplexing circuitry has to be calculated. This happens for example when we replace an adder by an ALU.

For registers we have the following possibilities:

1. A new register can be added.

2. An existent register that is not used at the moment can be used.

3. An existent register can be altered (expanding the bit width) and used.

By this strategy we are sure that we minimise the cost of the implementation. How well the overall realisation of the algorithm is depends on the fact how well the cost function is defined. When an optimum in the cost function is reached for an optimal implementation we will find this implementation. Therefore, it is very important to define the right cost functions.

5.7 **Hardware transformations.**

5.7.1 *Assumptions about the hardware.*

In this section we will describe the assumptions we have made about the hardware. These constraints are not inherent in strategy used in the silicon compiler but are an outcome of the current implementation of the hardware generation algorithm. They are not optimal but made a fast implementation possible.

1. All operator modules and register modules can be connected to each other.

2. Multiplexers can be added everywhere in between operator modules, register modules and operator and register modules.

3. Modules have to be available in the library to implement each node in the demand graph. Available means that a node can directly be mapped onto a module or that the function of the node can be realised by a set of modules.

4. The cycle time has to be long enough, to give the operator modules in the critical path, time to propagate the values to the registers where they have to be stored at the end of that cycle.

5.7.2 *Implementation of simple nodes.*

In these and the following sections some references are made to the description of a state. These concern the properties used to store some information needed during the implementation. A complete state description can be found in Appendix D.

In this section we deal with the implementation of simple nodes. Simple nodes are nodes whose implementation has no impact on the state machine. These nodes are constant nodes, put and get nodes and all operator nodes. We will explain their implementations in the following sections.

5.7.2.1 *Implementation of a constant node.*

The implementation is done as follows:

1. First is checked if the same constant is already implemented, if so the module that implements this constant is connected to the new use of this constant.

2. If the constant is not yet implemented a new module that implements this constant is made.

The constant module has the same outlook as a register in the hdw-list:

('Const constant-value (0) (outputs))

The value of the register is the value of the constant. The input of the constant module is net 0. The outputs are all nets that connect the module to all modules that use this constant.

5.7.2.2 *Implementation of an operator.*

In essence their are two possibilities for the implementation of an operator node.

1. A node can be mapped on an existing module, performing the same function and unused in the current cycle.

2. A new module can be made.

Both possibilities are investigated during the implementation of the operator node. All nodes that can be multiplexed are listed and for each node the cost of the additional circuitry is calculated. When there are already multiplexers at the inputs of the operator module they only have to be enlarged else multiplexers have to be added. The cheapest realisation is saved. Then the cost of adding a new module is calculated. When this is cheaper a new module is formed and else the cheapest multiplexing alternative is chosen. The hardware list of the new state is changed according to the previous decisions and all circuitry is connected to the right operators.

Monadic operators are treated in the same way as dyadic operators except that they have one input.

5.7.3 *Implementation of complex nodes.*

Complex nodes are nodes that affect the state machine. These nodes are the nodes that represent the control flow in the demand graph. These nodes have become part of the demand graph thanks to language constructs as IF ... THEN ... ELSE and WHILE ... DO. These nodes deliver new cycles in the state machine as described in the section about the control unit.

5.7.3.1 *Generation of new state machine cycles*

During the implementations within a cycle all information, concerning the used modules, is stored in the 'cycle-occ list. This information contains the module number and the signal value needed at the control input of the module during the cycle. When a new cycle has to be made we can use the 'cycle-occ list to generate the appropriate signals in the state machine

The actions performed during the closing of the previous cycle and the opening of a new one depend on the reason for a new cycle. Is it on account of a special language construct or on account of full occupation of the present hardware? We will first describe what happens in the latter case.

5.7.3.2 *Normal new cycle generation.*

In the 'cycle-occ list we find all operators that produce output during this cycle. All these outputs have to be stored. First, we try to fill all unused registers, present on the IC. When no more registers are available we add enough registers to store all values. Reusing registers means that some multiplexers in front of the registers have to be altered or to be placed.

The hardware used in this cycle is determined, thus we can expand the state machine to generate the signals to activate this hardware. Signals have to be made to all modules in the 'cycle-occ list and to the registers and multiplexers used to store the live variables.

After doing this a new cycle can be opened and the 'cycle-occ list be emptied.

5.7.3.3 *A new cycle on account of a conditional.*

We recognise a conditional in the pass of the demand graph when a set of *merge* nodes is encountered. Two new branches in a cycle have to be created. One in which the *then* branch nodes are implemented and another to implement the *else* branch nodes.

The old cycle is closed in the same way as a normal cycle is (see previous section). The cycle numbers of the two new created cycles are put in a 'special-struct-stack. If a node is implemented while the top of this stack is a list with two numbers, we know we have to determine in which cycle to put these node.

The hardware implementation of the merge nodes adds registers and multiplexers to the hdw-list .

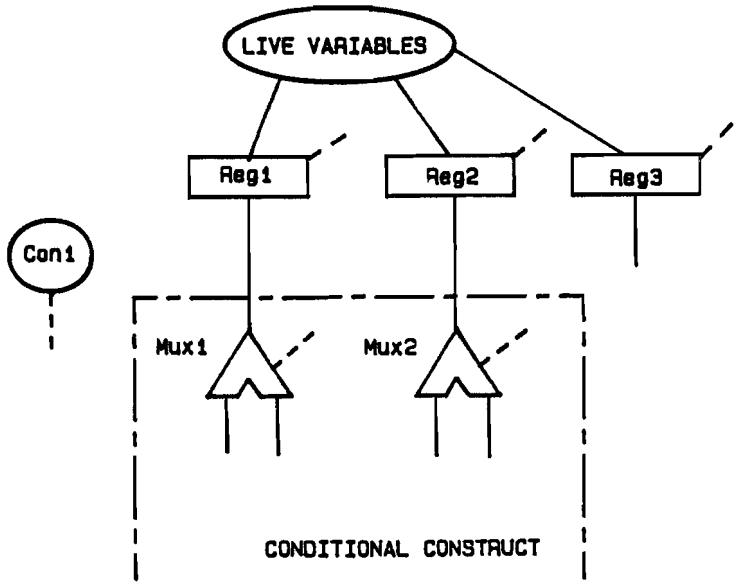


Figure 5.3. Implementation of merge-nodes.

The state machine looks like this:

```
( C1 ( ... )
     (Reg1.1) (Reg2.1) (Reg3.1)
     C2 )

( C2 (Con1.0)
     (Mux1.0) (Mux2.0) ... (Mux4.0) (Mux5.0) (Reg4.1) (Reg5.1)
     C3 )

     (Con1.1)
     (Mux1.1) (Mux2.1) ... (Mux4.1) (Mux5.1) (Reg4.1) (Reg5.1)
     C3 )

( C3    ...                                                    )
```

In cycle-1 the live variables are stored in the registers. (A 1 on the Reg. control port means: store the input). Cycle-2 directs the state machine to cycle-3 in two ways depending on the input from the Con1 module. The Con1 module delivers the test value for the conditional. The multiplexers, that control the data flow, will be signaled in this cycle too. We call this the selection cycle.

The conditional is ended when the corresponding set of *branch* nodes is encountered. The two data flows are directed into one flow. We therefore transform the *branch* nodes to multiplexers with multiple inputs. The outputs of these multiplexers are stored in registers by the normal store-live-variable procedure.

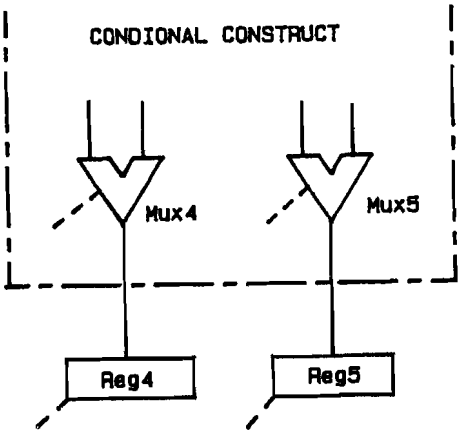


Figure 5.4. Implementation of branch nodes.

In cycle-2 the multiplexers Mux4 and Mux5 can be put in two states, to accept the data from the then branch or from the

else branch.

It is possible that there are more cycles between the cycle in which the cycles are separated and united again. For example the state machine can have the following outlook:

```
(C1     ...                          C2)
(C2     ...                          C3)
        ...   (Mux4.1)(Mux5.1)       C5)
(C3     ...                          C4)
(C4     ...   (Mux4.0)(Mux5.0)       C5)
(C5     ...
```

Now the then branch operators need three cycles and the else branch only one. In cycle C5 the cycles are united.

In the current implementation the registers Reg4 and Reg5 are implemented. It would be better not to add these registers but to wait until the need for a new cycle is reached in the process. The actions that are accumulated since the implementation of the branch nodes have to be done in both parts of cycle C2. (see first state-machine in this section). This strategy may save some registers and cycles in the state machine.

5.7.3.4 *A new cycle on account of a loop.*

Another structure that opens new cycles is the loop. A loop is entered through a set of *entry* nodes. When these nodes are encountered all live variables that are not input to the loop are saved in registers. The variables that are used in the loop are saved in registers that are preceded by multiplexers

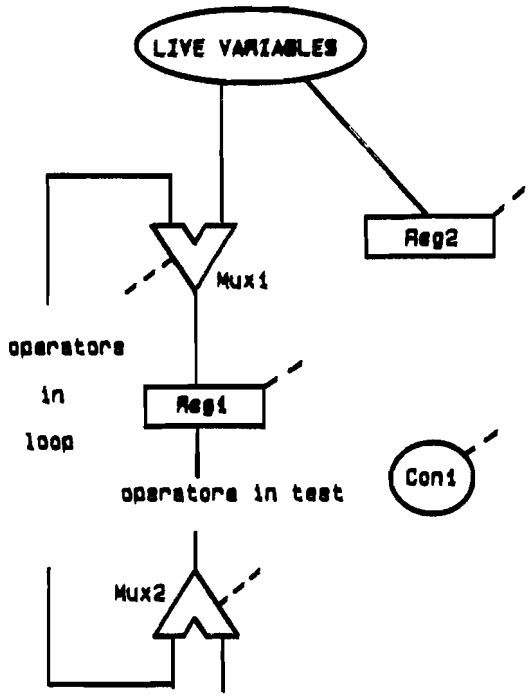


Figure 5.5. Implementation of a loop.

The cycle in the state machine is:

```
( C1 (     ...    )
     (Reg1.1) (Reg2.1) (Mux1.0)    ...
     C2 )
```

The values is entered in the loop through Mux1 and stored in Reg1. In cycle C2 the data is kept inside the loop: all multiplexers have value 1 on their control ports; or transported to the outside: Mux2 has value 0. In the first case the data is clocked in the register inside the loop again.

The state machine for these cycles:

```
( C2 (Con1.1)
     (Mux1.1)(Mux2.1) (Reg1.1) ...
     C2 )

     (Con1.0)
     (Mux2.0) ...
     C3 )

( C3     ...
```

The jump back to state C2 can take place in C2 itself or, if more cycles are needed to implement the operations within the cycle, in a later cycle. Another structure of the state machine possible when the implementation of the loop operations need one cycle more.

```
( C2 (Con1.1)
     (Mux2.1)  ...
     C2 )

     (Con1.0)
     (Mux2.0)  ...
     C4 )

( C3 ( ...)
     (Mux1.1)(Reg1.1)  ...
     C2 )

( C4 ...
```

5.7.3.5 *A new cycle on account of a procedure or function call.*

A function or procedure in the demand graph is called at least twice. Otherwise it would have been removed in the optimising step. Therefore we can implement the *param* nodes as a register with multiplexers in front of it. The number of inputs of the multiplexer is equal to the number of inputs of the *param* node. When a procedure is called, by the *call-in* nodes, we close the previous cycle and put the values in the appropriate registers. We jump in the state machine to the series of states that represent the procedure or function. The *result* nodes are implemented as registers with multiplexers behind them. In the last state of the procedure the result values are stored. The state machine for a complete procedure or function call:

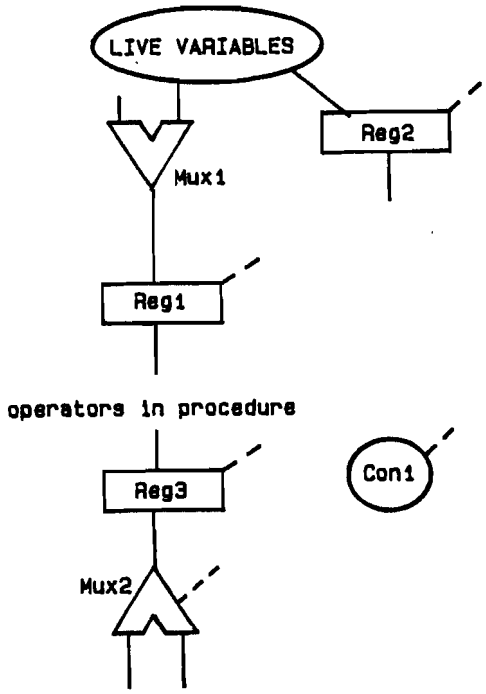


Figure 5.6. Implementation of a procedure.

```
( C1 ( ... )
     (procedure signals) (Reg3.1)
     C2 )

( C2 (Con1.0)
     (Mux2.0)
     C6 )

     (Con1.1)
     (Mux2.1)
     C? )
       :
( C5 (  ..  )
     (Mux1.0) (Reg1.1) (Reg2.1) (Con1.0) ...
     C1 )

( C6    ...
```

The number of states within the procedure is unlimited. The only constraint is that the next state of the last procedure state has to be the state following the calling state. When the procedure is called the call-id is stored in Con1. This is a register of width Log_2(number of calls). The last state of the procedure directs the state machine to the appropriate cycle, thanks to the value of Con1.

5.8 Example

The strategy explained in the previous sections is adapted to the GCD example. See the demand graph in the previous chapter. The call-in, param, result and call-out nodes are already removed in the optimising pass. This is because the procedure is called once. The resulting graph consists of two loops and some operator nodes. We describe the hardware generated for it: see fig. 5.7 and fig. 5.8.

The two put nodes are implemented by two input modules I1 and I2. They are activated in cycle S0 and store their values in R1 and R2 through M1 and M2. In cycle S1 a test is performed. Op1 delivers its value to the state machine which is directed to cycle S2 or S3. When R2=0 (OP1 delivers 0) we can output the value of R1 through M3 to the output module O1. Else we enter the second loop. While entering S2 the values of R1 and R2 are copied into itself. This is useless. A protection against the generation of such cycles or some postprocessing has to be added to avoid such cycles.

The value of register R2 is subtracted from R1, each time cycle S2 is traversed. When Op2 delivers 0 the values of R1 and R2 are exchanged and cycle S1 is entered again. This completes our description of the GCD machine.

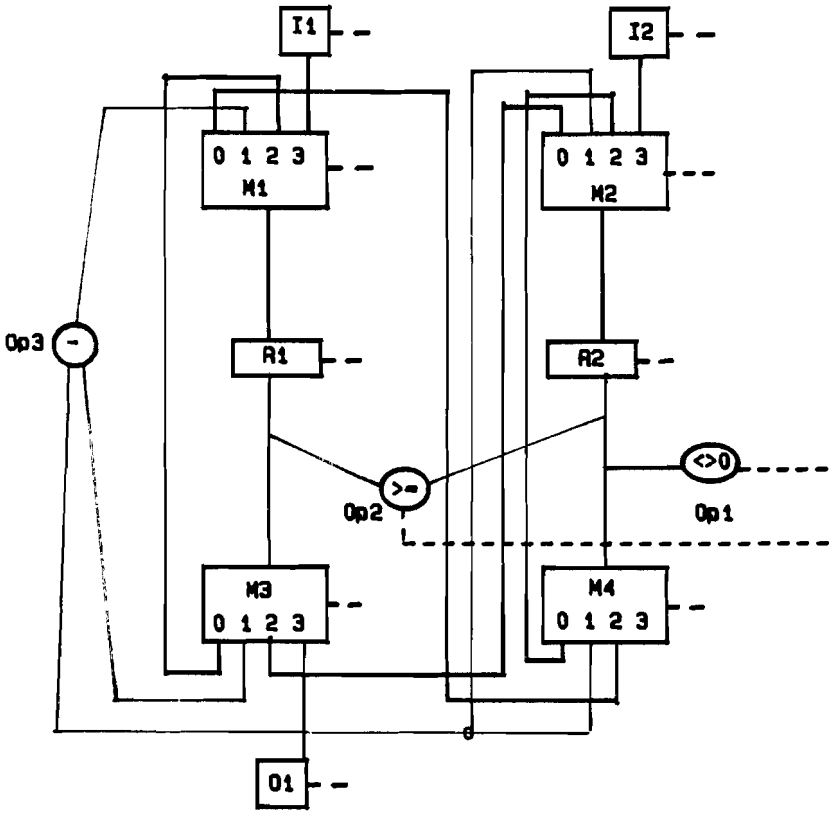


Figure 5.7. Hardware for the GCD machine.

```
(S0 { ... }
    {(I1.1)(I2.1)(M1.3)(M2.3)(R1.1)(R2.1)}
    S1)

(S1 {(Op1.1)}
    {(M3.0)(M1.2)(M4.0)(M2.2)(R1.1)(R2.1)}
    S2
    {(Op1.0)}
    {(M3.3)(O1.1)}
    S3)

(S2 {(OP2.1)}
    {(M3.1)(M4.1)(M1.1)(M2.1)(R1.1)(R2.1)}
    S2
    {(OP2.0)}
    {(M3.2)(M4.2)(M1.0)(M2.0)(R1.1)(R2.1)}
    S1)

(S3 END)
```

Figure 5.8. State machine for the GCD machine.

6. Conclusions and future research.

In this report a system is described to compile a behaviour description into a hardware description. The strategy used makes it possible to perform many optimisations during this compilation. The algorithms for demand graph construction, optimisations and hardware generation using dynamic programming are all coded in CommonLisp during the graduation period.

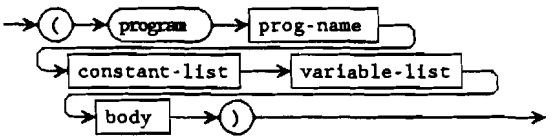
A lot of work has to be done to integrate the described system in a silicon compiler. First the language has to be defined and a parser made for it. Then the demand graph constructor can be expanded to transform all language elements. During the hardware generation more design alternatives can be generated. Of course the module library must be defined and the various costs for each module calculated. One of the main problems that remain is the definition of the cost functions. Probably this must be done by generating hardware for many behaviour descriptions and comparing these with the existent designs. The parameters in the cost functions can be altered to generate an optimal intergrated circuit. Last but not least the system must be interfaced with the lower levels of the silicon compilation.

Up till now necessary additions to the system are described. Furthermore we can develop other hardware generation mechanisms. The demand graph can serve as a basis and from here different strategies can be followed. An expert system or a mapping in stages are suggested. Mapping in stages means: first allocate the registers, next the operators and at the end the controller.

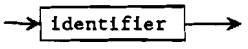
As indicated by the above suggestions, the current implementation is far from complete. But I hope that both the LISP implementation and this report will be useful in realising the silicon compiler in the near future.

Appendix A: Syntax tree

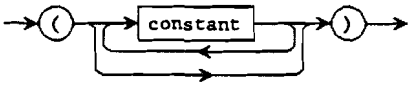
algorithm :



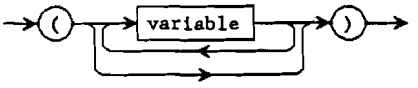
prog-name :



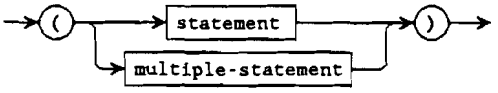
constant-list :



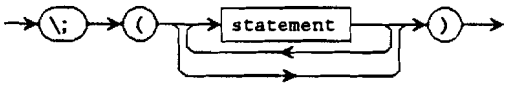
variable-list :



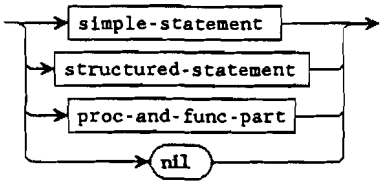
body :



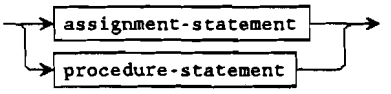
multiple-statement :



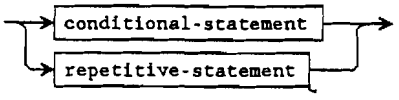
statement :



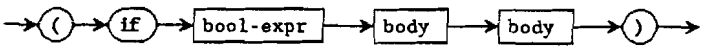
simple-statement :



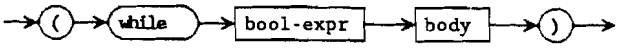
structured-statement :



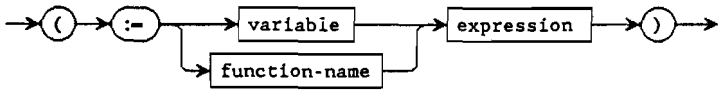
conditional-statement :



repetitive-statement :

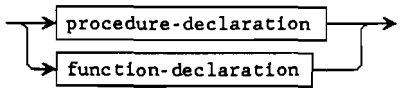


assignment-statement :

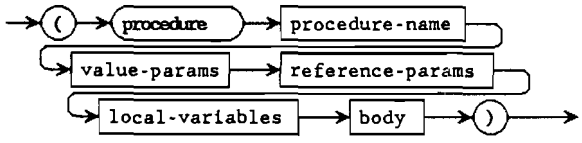


procedure-statement :

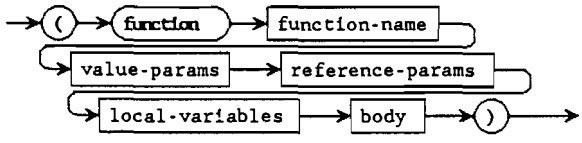
proc-and-func-part :



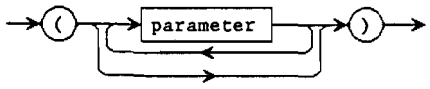
procedure-declaration :



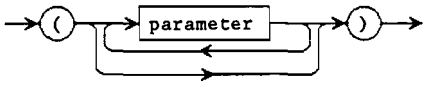
function-declaration :



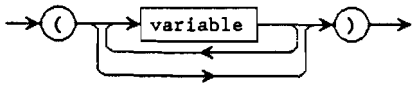
value-params :



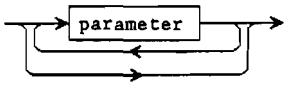
reference-params :



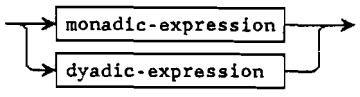
local-variables :



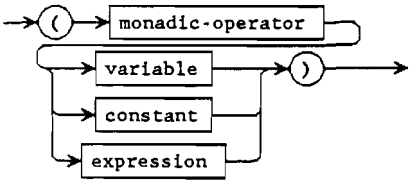
actual-parameter-list :



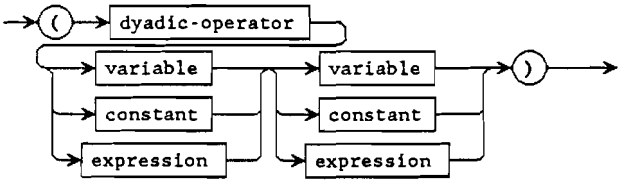
expression :



monadic-expression :

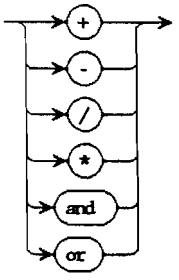


dyadic-expression :

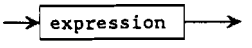


monadic-operator :

dyadic-operator :



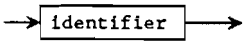
bool-expr :



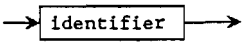
parameter :

procedure-name :

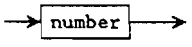
function-name :



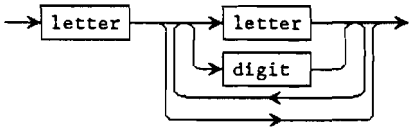
variable :



constant :



identifier :



Appendix B: Summary of used symbols with their properties.

These section is mainly included to provide some useful information for people, who have to deal with the demand graph, as constructed by my demand graph constructor.

In the demand graph constructor many symbols are used. The symbols can be put in classes. Depending on the fact in which class a symbol belongs it has some properties. Some properties are only used temporary and are not valid after the constructor has ended. They will not be discussed here. Other properties are still valid after the construction and can be used later. These are given in the following table. The criterion column gives the criterion for which a symbol is put in a class.

TABLE 7.1. Symbol classes used in the demand graph constructor.

Symbol Classes		
CLASS	CRITERION	PROPERTIES
<program-name>	assigned to the global variable *program-name*	'constant-list 'var-list 'procedure-list 'graph
<procedure-name>	in procedure list of the *program-name*	'outputs 'inputs 'value-params 'reference-params 'local-variables
<constant-name>	in constant list of the *program-name*	'value
<variable>	in varlist of *program-name* or in value-params, reference-params, local-variables of the actual procedure.	
<node-id>	prefix: "Node-" suffix integer	'type
<edge-id>	prefix: "Edge-" suffix integer	'type

In the following table are for each node-type given: all edge-types of edges that can leave that node-type, because the type of an edge is determined by the node it leaves. There are however a few exceptions to this rule. This is indicated in the table in the following way: when the type of the outgoing edge is determined by the node it enters then the name of the node it enters is given after the edge name in parenthesis (.). If a node determines the types of incoming edges these edges are given in curly braces {.} in the edge list.

TABLE 7.2. Node types with corresponding edge types

NODE-TYPE	EDGE-TYPE
sink	-
constant	source
dy-ops	left-source, right-source
mon-ops	source
and	left-source, right-source
or	left-source, right-source
link-in-0	inlink-success (merge) or last (exit)
link-in-1	inlink-failure (merge) or entry (exit)
entry	control, last, entry
exit	control, value, {last}, {entry}
branch	control, outlink-success, outlink-failure
merge	control, value, {inlink-success}, {inlink-failure}
param	proc-enter
result	value
call-in	value
call-out	proc-leave

Appendix C: Node treatment in the dynamic process.

For each node type is described when it is free and implementable and how the dynamic process is directed when the nodes are implemented. This is done by describing how the buckets of the states look before and after the implementation. Thus the bucket of the foregoing state is described, which is the bucket before the implementation, and the bucket of the new generated state. To treat the special construct nodes a stack is defined. The usage of the stack will become clear in the descriptions.

The descriptions are given in the following form:

Node-type : comment

```
(old-bucket)   -->   (new-bucket)
(old-stack)    -->   (new-stack)
```

The stack transform is omitted if the stack is not changed when the node is realised.

1. **Treatment of simple nodes.**

Sink,IO-sink:
Always free and implementable, implemented in State-0.

() --> (Node-0 Node-1)

Constant,dyop,monops,and,or:
Free if all outputs are implemented.
Implementable if free.

(bucket) --> (bucket minus implemented node)

2. Treatment of loops.

Loops are treated in a special way. A loop is entered through the entry node if all entry nodes belonging to the same loop are free. There is nothing implemented parallel to a loop. Thus the other nodes that were with the entry nodes in the bucket are pushed on stack. First the nodes inside the loop are encountered. When all exit nodes are freed, the inner-loop-nodes, freed by the exit nodes, are implemented first and the outer-loop-nodes are pushed on the stack. These nodes are implemented when the implementation of loop is finished. From the inner-loop-nodes the body of the loop is attached until a set with all entry nodes is reached again. Then the loop is finished and the process goes on with the implementation of all the other nodes from the stack.

entry (first time)
Free if node on 'entry edge implemented.
Implementable if all related entry nodes are free.

```
(entry-nodes | other nodes)      -->
     (free-nodes-after-entry)

(old-stack)                      -->
((entry-nodes | other-nodes) old-stack)
```

entry (second time)
Free if node on 'last edge implemented
Implementable if all related entry nodes are free.

It is determined that the entry nodes are attached the second time because the entry node set is on top of the stack.

```
(entry-nodes)                    -->
(other-nodes | free-nodes-after-link-in-1)

(old-stack)                      -->
          (cdr old-stack)
```

Remark:
Free-nodes-after-link-in-1 are the nodes freed by the link-in-1 nodes connected to the exit node of the loop. These link-in-1 nodes are put on the realised-nodes-list. Because the link-in-1 nodes are not essential to the hardware generation they are passed by in this way to delimit the number of generated states.

exit

Free if nodes on the 'control edge and the 'value are implemented.
Implementable if all related exit nodes are free.

```
          (exit-nodes)                    -->
            (in-loop-free-nodes-after-exit)

          (old-stack)                     -->
 (cons (car old-stack link-in-1-nodes)(cdr old-stack))
```

Remark:
The in-loop-free-nodes-after-exit are the nodes that are freed by the link-in-0 nodes connected to the exit nodes. These link-in-nodes are passed by, by immediately putting them in the realised nodes list. The link-in-1 nodes are pushed on the stack in the same list as the other-nodes. These are the nodes that have to be put in the bucket when the loop is completely implemented.

3. Treatment of conditionals.

merge

Free if nodes on the 'control edge and the 'value are implemented,
Implementable if all related merge nodes are free.

(merge-nodes | other-nodes) -->
(free-nodes-after-link-nodes | other nodes)

Remark:
The merge nodes and the connected link-in nodes are put in the realised nodes set. The nodes that are freed by the link-in-1 and the link-in-0 nodes, together with the other nodes are put in the new bucket.

Branch

Free if nodes on all the outgoing edges are implemented.
Implementable if all related branch nodes are free.

(branch-nodes | other-nodes) -->
(free-nodes-after-branch-nodes | other nodes)

Remark:
The branch nodes are put in the realised nodes set. The nodes that are freed by the branch nodes together with the other nodes are put in the new bucket.

4. Treatment of procedures and functions.

Procedures are treated in a way similar to loops. No other nodes are implemented during the implementation of the procedure. The other nodes are pushed on a stack that is popped when the result nodes are treated. When a procedure is called a second time it is already implemented in hardware. Thus the new bucket is formed with the result nodes and the other nodes are put on the stack. The preceding implementation is used again.

Call-in (first time)

Free if nodes on all the outgoing edges are implemented.
Implementable if all related call-in nodes are free.

```
(call-in-nodes | other-nodes)   -->
   (free-nodes-after-param-nodes)

(proc-stack)                    -->
(cons (proc-call-node other-nodes) proc-stack)
```

Remark:
The other nodes are pushed on the stack together with the proc-call node. The proc-call node is used to detect to which call the results have to be sent when the result nodes will be implemented. In the new bucket are the nodes freed by the param nodes. To delimit the number of states these param nodes are passed by.

Call-in (second time)

Free if nodes on all the outgoing edges are implemented.
Implementable if all related call-in nodes are free.

```
(call-in-nodes | other-nodes)   -->
            (result-nodes)

(proc-stack)                    -->
(cons (proc-call-node other-nodes) proc-stack)
```

Remark:
The other nodes are pushed on the stack together with the proc-call node. The proc-call node is used to detect to which call the results have to be sent when the result nodes will be implemented. The result nodes are put in the new bucket to avoid a new implementation of the procedure.

Result

Free if nodes on all the outgoing edges are implemented.
Implementable if all related result nodes are free.

```
          (result-nodes)                    -->
    (free-nodes-after-call-out-nodes | other-nodes)

         (proc-stack)                       -->
                   (cdr proc-stack)
```

Remark:
In the new bucket are put the other nodes and the free nodes after the call-out nodes. In this way the call-out nodes are passed by and delimits this strategy the number of generated states. The call-out nodes are put in the realised nodes set.

Appendix D: State description.

In each state enough information must be stored to be able to determine the next states in an optimal way. Characters in between "(" and ")" are only provided to explain the property name but are omitted in the implementation. The following properties are stored for each state (if necessary):

1. **Cost**: the cost of the implementation up to his state, updated when new hardware is implemented.

2. **Bucket**: the nodes that are free when entering this state, updated when node(s) are implemented.

3. **Transform-list**: list of pairs of nodes that are realised and whose output is still needed by other nodes that have not been implemented and the module number of the module in which the node is compiled. Pairs are entered when a module is made for a node, pairs are removed when a result of a node is used the last time.

4. **Cycle-occ(upancy)**: the occupancy of the current processor cycle, updated when modules are used, cleared when a new cycle is entered.

5. **Special-cycle-occ**: the occupancy of the second current processor cycle. Used when conditionals are implemented and operators for two cycles are collected.

6. **Loop-cycle-occ**: the occupancy of the processor cycle in a loop.

7. **H(ar)dw(are)-list**: the generated hardware on the integrated circuit.

8. **Stat(e)-mach(ine)**: the state machine to control the generated hardware, updated each time a new cycle is made.

9. **Input-signal**: contains pair of net number and value of the net that is the input signal of the *then* branch cycle. Updated when the test node of the if-statement is implemented. Cleared when a new cycle is made.

10. **Special-input-signal**: same as input-signal only for *else* branch cycle.

11. **Loop-input-signal**: same as input-signal only for loop cycle, when loop is entered from normal environment or from *then* branch environment. Updated when the node that delivers the test signal of the while-statement is implemented.

12. **Loop-special-input-signal**: same as loop-input-signal only for loop cycle, when loop is entered from *else* branch environment.

13. **Insert-state**: When a loop is ended and the environment surrounding it is entered, the insert-state is the cycle from which the jump out the loop must be made. This property is stored when entering a loop and deleted when a loop is ended.

14. **Special-struct-stack**: Top of stack represents the environment that is entered. Two cycle numbers on top of stack mean that an if-environment is handled, one cycle number that a loop is handled. The cycle numbers are the cycles in which the special structure is entered. If more states are needed to implement the operators in a special structure the cycle number is replaced by the new cycle number.

References

[Aho86] Aho A.V., Sethi R., Ullman J.D.,
"Compilers principles, techniques and tools",
Addison-Wesley publishing company,1986

[All70] Allen, F.,
"Control flow analysis",
SIGPLAN Notices 5, 1-19 (1970).

[Babi78] Babich, W.A. and Jazayeri, M.m
"The method of attributes for data flow analysis",
Acta informatica 10, 245-264, 1978.

[Bell57] Bellman R.E.,
"Dynamic programming",
Princeton University Press, Princeton New Jersey, 1957.

[Bell62] Bellman, R.E., Dreyfus S.E.,
"Applied dynamic programming",
Princeton University Press, Princeton, New Jersey, 1962.

[Black85] Blackburn R.L., Thomas D.E.,
"Linking the behavioural and structural domains of representation in a synthesis system",
Proc. 22nd Design Automation Conf., 1985.

[Camp85] Camposano. R,
"Data path synthesis from behavioural level descriptions in DSL."
in: VLSI Algorithms and architecture,
Elseviers Science Publishers,
North-Holland, 1985.

[Duley68] Duley J.R. and Dietmeyer D.L.,
"A Digital System Design Language.",
IEEE Trans. Comput. Vol C-17, No. 9,
PP. 850-861 (1968)

[Hitch83] Hitchcock III C.Y., Thomas D.E.,

"A method of automatic data path synthesis",
Proc. 20th Design Automation Conf., June 1983.

[Kenn81] Kennedy, K.,
"A survey of data flow analysis techniques",
in:
S.S. Muchnick, N.D. Jones,
"Program flow analysis: theory and applications",
Prentice-Hall, Inc., Englewood Cliffs,
New Jersey, pg. 5-54, 1981.

[Mano85] Mano. T. et al.,
"OCCAM to CMOS."
in: CHDL85
Elsevier Science Publishers B.V.,
North-Holland, 1985.

[Rose77] Rosen, B.K.,
"High level data flow analysis",
Comm. of the ACM, Vol. 20, Numb. 10, Oct. 1977.

[Rosen84] Rosenstiel. W,
"Synthese des Datenflusses digitaler Schaltungen aus formalen Funktionsbeschreibungen",
Fortschritt-Berichte der VDI-Zeitschriften,
Reihe 10, nr. 37,
VDI-Verlag GmbH, Dusseldorf 1984.

[Rosen85] Rosenstiel. W and Camposano. R,
Synthesising circuits from behavioural level specification."
in: CHDL85 PP.391-403
Elsevier Science Publishers B.V.,
North-Holland, 1985.

[Thom83] Thomas D.E. et al.,
"Automatic data path synthesis",
Computer, p. 59-70, December 1985.

[Veen85] Veen, A.H.,
"The misconstrued semicolon",
Dissertation, Eindhoven University of Technology,
CWI, Amsterdam, 1985.