ZEUS

A hardware description language for VLSI

ETH

Eidgenössische Technische Hochschule
Zürich

Institut für Informatik

Karl J. Lieberherr, Svend E. Knudsen

ZEUS: A HARDWARE DESCRIPTION LANGUAGE FOR VLSI

February 1983 51

Adresses of the authors:

Prof. Karl J. Lieberherr
Princeton University
Dept. of Electrical Engineering & Computer Science
Princeton, New Jersey 08540, USA

Svend E. Knudsen
Institut für Informatik
ETH-Zentrum
CH-8092 Zürich, Switzerland

Zeus: A Hardware Description Language for VLSI

Karl J. Lieberherr *

Institut für Informatik ETH Zürich
and
Department of Electrical Engineering and Computer Science
Princeton University

Svend E. Knudsen

Institut für Informatik ETH Zürich

Abstract

A technology-independent, simple, but powerful hardware description language is proposed. The language is suitable for describing VLSI algorithms from the architecture to the logical level. The layout language part which is suitable for any two dimensional technology, allows the user to describe the relative positions of the components and pins of the layout as well. With technology-dependent language extensions, which are the subject of another paper, the Zeus text can be refined down to the transistor level.

Zeus is a functional and structural design language. A Zeus program consists of a sequence of constant, type and signal declarations. The structure of a hardware component is described by a component type definition. A component type T is instantiated by declaring a signal S of component type T. We call S a signal since we consider a component to be represented by its interface, i.e. the signals which enter and exit.

Signals (e.g. components) are connected by connection and assignment statements. The basic signal types are boolean and multiplex (tri-state) but signals can be structured as nested arrays and records. A signal of type boolean is assigned a logical value exactly once, while a signal of type multiplex may be assigned conditionally several times. These and other rules are checked at compilation and simulation time to prevent the "burning" of transistors and to exclude other mistakes.

The language has been tested on a variety of examples like: finite state machines, multiplexors, adders, pattern matching, AM2901, dictionary machines, systolic stacks and it is currently in the process of being implemented.

---

* Partially supported by National Science Foundation under grant MCS80-04490. Oct.82

Contents

1. Introduction

Computer scientists and engineers have continuously invented and employed notations for modeling, specifying, simulating, documenting, communicating, teaching and controlling the design of digital systems through the three decades of digital computer history. Initially electronic systems were represented via circuit schematics. Following Shannon's revelation of 1938, logic diagrams

and/or boolean equations represented digital systems in a fashion that deemphasized electronic and fabrication detail while revealing logical behavior. As system complexity grew, block diagrams, timing charts, sequence charts, other graphic and symbolic notations were found to be useful in summarizing the gross features of a system and describing how it operated. In addition, it always seemed necessary or appropriate to augment these documents with lengthy verbal descriptions in a natural language.

While each notation was, and still is, a perfectly valid means of expressing a design, lack of standardization, conciseness, and formal definitions interfered with communication and understanding. This problem was recognized early and formal languages began to evolve in the 1950's. Read [Read(1952, 1953)] developed a notation that became known as a register transfer language. While this notation had only a few of the features that are associated with register transfer languages (RTLs), its development started an evolutionary process that is still underway.

Programming language development influences RTL evolution. Because high level programming languages were developed more rapidly and studied far more extensively, RTL syntax has been influenced by known programming languages. In some cases [Robinson(82)] programming languages were proposed with or without slight modification as RTLs so that simulation may be performed via available software.

Reed's RTL, and the at least three dozens that followed (e.g. [INMOS(82), Lattice(82), Wirth(82)]), have been used in many ways, often to enhance other notations rather than replace them. RTL descriptions of digital systems are used to communicate such information to computer programs which simulate the described system or translate (semi-automatically) the description (for example to TTL wiring lists or MOS layouts [Siskind (1982)]). Thus RTLs might be rated according to their effectiveness as communication tools and are a crucial part of design automation systems.

We propose a new notation, called Zeus, which was inspired by brother Hades [Wirth (1982)]. Zeus has almost the expressive power of the well-known register transfer language DDL [Duley (1968)], however Zeus programs appear to be more readable and offer more security by compile time checks. Hades and Zeus don't allow every possible circuit to be expressed. We disallow feedback loops which do not lead through registers and we restrict the legal assignments to signal variables in several ways. This should be an advantage, because it may prevent designers from critical designs, simplify simulation and preclude errors that are difficult to pinpoint. The semantics of Zeus imply a simulator which is conceptually simpler than state-of-the-art switch-level circuit simulators [Bryant (1981), Lipton (1982), Mehlhorn (1982)]. Both Hades and Zeus are suitable for describing systolic algorithms. In Zeus the activities of each beat are represented by a sequence of assignments, function component calls and connection statements which determine how the signals are propagated, manipulated and stored into registers.

Zeus unifies the Hades-concepts of a block and a module into one concept: a component type. A component type has a parameter list like a Hades-module. While the Hades-module parameters can be influenced only by a module call, the Zeus-component type parameters can be accessed like the fields of a Pascal record. Combined with the Zeus connection statement this allows a succinct description of interconnections without auxiliary variables. In contrast to Hades, Zeus defines a piece of hardware as a component type which is instantiated by a signal declaration. The parameters of a component type can be special component types themselves. In Zeus, signals can be structured like Pascal variables with the array and record construction. A Zeus-record type is simply a component type without body (no internal connections). Hades only allows arrays of a simple type.

Some aspects of the structural part of Zeus have been developed independently at MIT [Lim (1982)]. The MIT language which is called HISDL uses components and specifies connections in a similar way as Zeus. However HISDL is intended only for a structural description. In section 4 of this paper we translate the HISDL specification of a routing network given in [Lim(1982)] to Zeus.

Some aspects of the functional part of Zeus have been developed at USC [Hayes (1982)] in form of the CS theory. However, intentionally, not every CS network can be expressed as a Zeus component. This will be possible in a MOS-technology-dependent language extension of Zeus.

Leiserson and Saxe [Leiserson/Saxe (1981)] use the same model for synchronous ircuits as Zeus and they provide an elegant design methodology for developing ertain systolic Zeus programs. [Cappello (1982)] and [Moldovan (1982)] contain urther results which can be applied to the analysis and synthesis of Zeus programs.

The order statement of the layout language is taken from [Valdes(82)].

The paper is organized as follows: Sections 2-8 define the language. Section 7 contains the complete, cross-referenced syntax in extended Backus-Naur form (see e.g. [Wirth(82)]). Sections 1 and 9 relate Zeus to previous work and section 10 contains several examples of Zeus programs.

2. Vocabulary and representation

The vocabulary of Zeus consists of identifiers, numbers and special symbols. Identifiers are used to denote the objects of a circuit.

Numbers may be followed by a letter B or b to specify an octal number.

Special symbols are the following sequences of characters.

```
+-              plus minus
()[]            parentheses
.,;:            punctuation
< <= > >= =     relation
:= ==           assignment
```

```
..          range
*           unspecified, multiplication
<* *>       comment
```

AND ARRAY BEGIN BIN BOTTOM CLK COMPONENT CONST DIV DO DOWNTO ELSE ELSIF END FOR IF IN IS LEFT MOD NOT NUM OF OR ORDER OTHERWISE OTHERWISEWHEN OUT PARALLEL RSET RESULT RIGHT SEQUENTIAL SEQUENTIALLY SIGNAL THEN TO TOP TYPE USES WHEN WITH

3. Declarations

Declarations in general serve to introduce objects into the program and to associate an identifier with the object. The identifier is valid within the component type in which the declaration occurs. All constants, types and signals must be declared before they are used. Signal declarations must occur after the constant and type declarations. These rules imply that non-local signals (except a predefined clock and a predefined reset signal) are not allowed in Zeus.

3.1 Constant declarations

A constant declaration associates an identifier with a numerical or a signal constant. Signal constants are nested arrays and records of the basic types boolean and multiplex (defined later); the structure is shown with parenthesis. The basic signal constants are 0,1,UNDEF (undefined) and NOINFL (no-influence (disconnected, high-impedance)) which will be explained later. The type of 0,1 and UNDEF is boolean; the type of NOINFL is multiplex.

Examples

```
start=(0,0,0) ; a=((0,1),(1,0),(0,0)) <* signal constants *> ;
length = 7  <* numerical constant *>
```

For numerical constant expressions we have adopted the Modula-2 syntax.

Syntax

```
constDeclaration = CONST { ident "=" constant ";" } .
 ident = letter { letter | digit } .
 constant =ConstExpression | sigConstExpression.
  ConstExpression = SimpleConstExpr [ relation SimpleConstExpr ] .
   relation = "=" | "<>" | "<" | "<=" | ">" | ">=" .
   SimpleConstExpr = "=" ["+"|"-"] ConstTerm
    { AddOperator ConstTerm } .
   AddOperator = "+" | "-" | OR .
   ConstTerm = ConstFactor { MulOperator ConstFactor } .
   MulOperator = "*" | DIV | MOD | AND .
   ConstFactor = number | "(" ConstExpression ")" | NOT ConstFactor |
   ident [ "(" ConstExpression { ";" ConstExpression } ")" ] .
```

```
  number = digit { digit } ["B" | "b"].
sigConstExpression = "(" sigConstExpression
  { "," sigConstExpression } ")" |
value | BIN "(" ConstExpression "," ConstExpression ")".
  value = "0" | "1" | ident.
```

3.2 Type declarations

Zeus provides COMPONENT, ARRAY and basic types which can be arbitrarily nested. A component type defines a circuit with its input and output signals as parameters. Internal signals and functions are declared as local objects. A component type is instantiated by a signal declaration; the component type definition merely defines the circuit pattern. (Component type declarations may be viewed as corresponding to type declarations in programming languages, whereas component instantiations correspond to variable declarations.)

Example

```
TYPE
 halfadder =
 COMPONENT (IN a,b: boolean; OUT cout,s: boolean) IS
  BEGIN s := XOR(a,b);
    cout := AND(a,b)
  END;
```

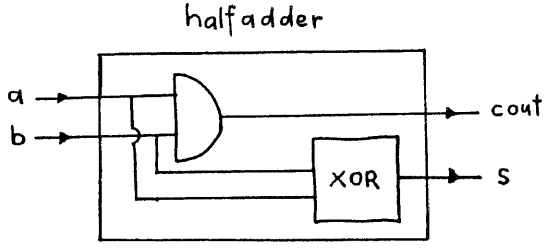


Fig. 3.2.1

```
fulladder =
COMPONENT (IN a,b,cin: boolean; OUT cout,s: boolean) IS
 SIGNAL h1,h2:halfadder;
```

```
BEGIN h1(a,b,*,h2.a); h2(h1.s,cin,*,s);
 <*the * indicates that no connection is made*>
 cout := OR(h1.cout,h2.cout)
END;
```

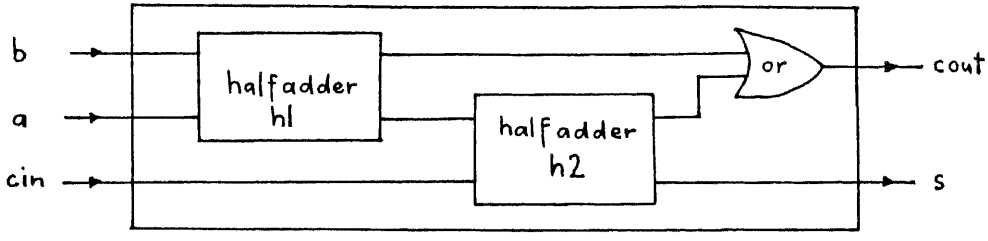


Fig. 3.2.2

A function component type is a component type which returns a value. A function component type declaration is instantiated by a call. Function component types cannot be used in signal declarations.

Example

```
TYPE
bo(n) = ARRAY[1..n] OF boolean;
mux4 = COMPONENT ( IN d:bo(4); IN a:bo(2); IN g: boolean ) :boolean;
 CONST bit2 = ( (0,0),(0,1),(1,0),(1,1) );
 SIGNAL h: multiplex ;
 BEGIN
  FOR i:=1 TO 4 DO IF EQUAL(a,bit2[i]) THEN h :=d[i] END END;
  RESULT AND(NOT g,h)
 END;
```

Local declarations are valid only within the component type in which they occur. The type identifier of a component type is used in signal declarations and calls (in case of a function component type). The value of a function component type is specified by the expression following the symbol RESULT. An assignment to the type identifier is illegal.

The parameters of a component type declaration are called formal parameters. They are either IN, OUT or INOUT parameters. An IN (OUT) parameter is used to transmit a value to (from) a component. An INOUT parameter is specified by

omitting the IN and OUT reserved word and it is used for communication to and from the component.

For the purpose of connecting signals, the formal parameters of an instantiated component can be accessed with a notation as for the fields of a record in Pascal (forget the IN, OUT indicators). However the connections to a function component can be specified only in a function call. If a structured formal parameter is an INOUT parameter it may be that some substructures are either IN or OUT parameters. The IN or OUT property is inherited by substructures. A substructure may not be at the same time an IN and OUT parameter. A parameter X of a component type Y can be a component type itself.

The "uses" list of a component type A names all objects (constants or types, but not signals) which are defined outside A but used inside A. If the uses list is omitted then every object defined in the environment of A can be used in A. If the uses list is empty then nothing from the outside is used in A. Predefined standard types (e.g. the function component types AND, OR, NAND ... and the component type REG) are pervasive and can be used everywhere without mentioning in a uses list.

The parameters of a function component type are just place holders as the parameters in a programming language like Pascal or Modula-2. These parameters can be renamed in the type definition without affecting the correctness of the program. However the renaming of the parameters in a component type definition only, usually changes the meaning of the program. It might be necessary to rename the names also outside the type definition in case the parameters are accessed like the fields of a Pascal record. Therefore the parameter names are an important part of the interface of a component type. If a component type is imported in a uses list, all its parameter names are imported too.

The elements of the statement sequence define the functional interconnections of the various signals. The order of the statements is irrelevant, but all assignees must be well defined. This implies that no loops must occur in the definitions of signal values. However loops leading through registers are allowed (see section on storage elements).

A component type without body represents a record type of signals (in the Pascal sense). Such a record type represents a sequence of signals (wires) where each might have an individual type. The IN and OUT indicator may also be used in such record definitions.

Example

```
TYPE
bus = COMPONENT ( r,s,t : bo(3) ; u : boolean);
```

An ARRAY type represents a sequence of signals, all of the same type.

Example

TYPE
bus4 = ARRAY [1..4] OF bus ;

The basic types are "boolean" and "multiplex". Signals of type multiplex are tri-state signals which can have the high-impedance value NOINFL. We prefer the name multiplex to tri-state.

For signals of basic type the following rule holds (with two exceptions): Either they are unconditionally assigned a value exactly once (syntactically) or they are assigned a value an arbitrary number of times through an if statement.

Example

IF EQUAL(a,b) THEN r:=s END; t:=f(a,b)

r is assigned a value through an if statement. If all RESULT statements of f are contained in if statements (i.e. f is of type multiplex) then t is also assigned a value through an if statement.

The two exceptions are IN parameters of an instantiated component and formal OUT parameters. Such parameters of a basic type must be boolean (see below), however they can be treated like signals of type multiplex : Variables of a basic type which are assigned through an if statement have to be either of type multiplex or they have to be an IN parameter of an instantiated component or a formal OUT parameter which is used inside the defining component type. In the last two cases an implicit multiplex signal is generated and assigned to the IN or OUT parameter.

Unstructured IN and OUT parameters must be of type boolean. INOUT parameters of a basic type must be of type multiplex. This rule implies that no assignments are allowed to a formal IN parameter within the defining component type and that no assignment is allowed to an OUT parameter of an instantiated component type.

The simulator checks that at most one (0,1,UNDEF)- assignment takes place at "runtime",i.e. if a signal is assigned conditionally through an if statement then it has to be guaranteed that during the operation of the chip at most one (0,1,UNDEF)-assignment is active. This rule safeguards against "burning" transistors.

The unconditional assignment x:=y with x of type multiplex and y of type boolean is legal. However no further assignments (conditional or unconditional assignments) are allowed to x. In an assignment x:=y it is allowed that x is of type boolean and y of type multiplex. It is assumed that hardware (e.g. an amplifier) will be generated automatically to perform the type conversion.

These rules extend in a natural way to signals of structured type. All basic substructures must satisfy the assignment rules.

Types can be parameterized with integers in Zeus. The formal parameters of a

type definition (to the left of "=") are valid in that definition only. The actual parameters are numerical constant expressions and they are specified when types are instantiated in signal declarations. Parameterized types are especially useful in connection with recursion.

The actual parameters of a function component are specified in brackets in a call statement. For example, plus[n](a,b) is the call of a function component type which will return the sum of the two n-bit numbers a,b.

Syntax

```
typeDeclaration = TYPE { ident [ "(" idlist ")" ] "=" type ";" }.
  type = arrayDeclaration | componentDeclaration |
    ident ["("ConstExpressionList")"].
  arrayDeclaration =
    ARRAY "[" ConstExpression ".." ConstExpression "]" OF type .
  componentDeclaration =
    COMPONENT "(" [ fparams { ";" fparams } ] ")"
     [ "{" layoutStatementList "}" ]
     [ [ ":" type ] IS [ USES idlist ";" ] { declaration }
     [ "{" layoutStatementList "}" ] BEGIN StatementSequence END ] .
    fparams = [ IN | OUT ] fieldlist .
     fieldlist = idlist ":" type .
     idlist = ident { "," ident } .
```

.3 Signal declarations

Signals in Zeus correspond to variables in programming languages. A signal may be structured with the COMPONENT and ARRAY facilities. Signals of type multiplex have values 0,1,UNDEF (undefined),NOINFL (no-influence) and signals of type boolean can have values 0,1 and UNDEF. NOINFL is the disconnected or high-impedance state. An instantiated component type is called a signal since the interface of a component consists of a sequence of signals.

Syntax

```
signalDeclaration =
  SIGNAL { idlist ":" type [ "(" ConstExpressionList ")" ] ";" } .
```

4. Statements

In contrast to Pascal-like languages, the relative order of statements does not influence the semantics of a Zeus program.

Syntax

```
StatementSequence = statement { ";" statement } .
  statement = [ assignment | replication | condGeneration | connection |
```

conditional | result | parallel | sequential | with] . (*empty*)

4.1 Assignments

Assignments denote signal definitions and connections. The assignment s:=e signifies that signal s be defined by the expression e. The "direction" of the equal sign to the left indicates the intended "signal flow". We require that the type of e has the same number of substructures of basic type as the type of s.

There is a second kind of assignment statement which is used for aliasing signals. An aliasing operation x==y connects the two signals. The consequence is that we have one signal with two (or more) names. It is required that x and y have the same number of basic substructures and that they are pairwise of type multiplex. There are two exceptions: One of the basic signals x or y may be of type boolean, if it is an IN parameter of an instantiated component or a formal OUT parameter. In case of the exceptions there will be an implicit type conversion from multiplex to boolean. If a signal of type boolean is assigned with "==" then it may not unconditionally be assigned with ":=". An aliasing operation x==y must not occur within a conditional statement.

If both x and y are signals of type multiplex then the assignment x:=y is illegal. x==y has to be used instead. If x is a variable of type boolean then the assignment x:=NOINFL is replaced by x:=UNDEF.

If the assignee s is of type boolean and not an IN parameter of an instantiated component or a formal OUT parameter used inside the defining component type then exactly one unconditional assignment must occur. Several conditional assignments may occur to signals of type multiplex, to a boolean IN parameter of an instantiated component or to a boolean formal OUT parameter. (The type conversion from multiplex to boolean will be done automatically.) A variable may not be assigned conditionally and unconditionally.

Expressions (and statements) describe the functional composition of a circuit. The value of a function component call is defined by the corresponding type declaration and the values of the actual parameters. The type of a function component type cannot contain a component type with a body, i.e. it is a nested type of records and arrays of a basic type.

The following are standard function component types; they are implicitely defined and require no declaration:

AND(x0,x1, ... ,xn)
OR(x0,x1, ... xn)
NAND(x0,x1, ... ,xn)
NOR(x0,x1, ... ,xn)
XOR(x0,x1, ... ,xn)
NOT x
EQUAL(x0,x1)

For the AND,OR,NAND,NOR,XOR and EQUAL function we require that x0,x1, ... ,xn have the same number m of substructures of basic type. The result type is an ARRAY[1..m] OF boolean. The result type of NOT is an ARRAY[1..m] OF boolean where m is the number of substructures of x of a basic type. The operations are performed bit-wise. The detailed definition of these functions is given in the semantics section.

The standard function BIN(a,b) transforms the numerical constant a into ARRAY [1..b] OF boolean.

The standard function NUM(s) transforms a signal s into a numerical value. This function is for example useful for defining a multiplexor or a random access memory.

A signal which is of a structured type denotes all subcomponents of basic type in natural order if a selector is omitted.

Examples

SIGNAL score: ARRAY [1..5] OF boolean;

In the statement part score denotes the five signals score[1], score[2], ... ,score[5].

SIGNAL r:ARRAY[1..n] OF COMPONENT (IN in:boolean; OUT out:boolean);

r.in denotes the n basic signals r[1..n].in.

This rule might lead to ambiguous situations.

Example

SIGNAL matrix: ARRAY[1..n] OF ARRAY[1..n] OF boolean;

matrix[2] might denote the second row or the second column of the matrix. For such cases we define the default that right-most selectors have been omitted first. Therefore matrix[2] is equivalent to matrix[2][1..n].

A sequence of signals may be assigned in an assignment statement. For example, if x is an ARRAY[1..n] OF boolean then x[2..7] denotes the six boolean signals x[2], x[3], ... ,x[7]. A similar construction is possible with the component constructor (see syntax definition).

Let s be a signal which is an instantiated component local to another component C. If a port s.b of s is neither used nor assigned in C (also not with a connection statement) then an error message is generated, provided that at least one port of s is used or assigned. In other words, in Zeus it is legal to have completely disconnected components (see the next subsection about conditional generation), but unused ports of relevant (i.e. not completely disconnected) components have to be "closed" explicetly with an assignment or connection statement. "*" has the meaning of "empty signal" or "no connection".

Example

```
COMPONENT(....) IS
TYPE
 atype=
 COMPONENT(IN x:boolean;
       OUT y:boolean; z:multiplex) IS
 BEGIN
 ...
 END;
SIGNAL
 g: ARRAY[1..2] OF atype;
 x1,x2: boolean; x3:multiplex;
BEGIN
 ...
 g[1](x1,x2,*); <* see the section "4.3 Connections" *>
 <* equivalent to
   WITH g[1] DO
    x:=x1; x2:=y; z==*
   END;
 *>

 WITH g[2] DO
   x:=*; *:=y; z==x3
 END
END;
```

In the following we summarize the rules for "*". Intuitively "*" represents the empty signal.

Let x be an instantiated component and x.b a port of type boolean. The assignment "x.b:=*" is considered to be an empty assignment (like no assignment) to x.b. If no other (non-empty) signal is assigned to x.b then "x.b:=*" is equivalent to x.b:=UNDEF. After the assignment "*:=x.b" the signal x.b is still available. (The assignment "*:=x.b" might happen with a connection statement since unknown actual parameters are represented by "*", see section 4.3.)

If b is a port of type multiplex then "x.b==*" (or "*==x.b") is considered to be an empty assignment (like no assignment) to x.b. If no other (non-empty) signal is assigned to x.b then "x.b==*" is equivalent to x.b==UNDEF.

"*" is typeless and on the left-hand-side of an assignment statement it may be assigned conditionally or unconditionally several times (with ":=" or "==").

Syntax

assignment = signal (":=" | "==") expression .

```
signal = ( ident { "[" (ConstExpression
 [ ".." ConstExpression ] "]"| NUM "(" signal ")" ) "]"
 | "." ident [ ".." ident ] } ) | "*".
expression = signal | functionComponentTypeIdent
   ["("ConstExpressionList")"]
   [ expression ] | BIN "(" ConstExpression ","
   ConstExpression")" |
   sigConstExpression | "*" [ ":" ConstExpression ] |
   "(" expression { "," expression } ")" .
 functionComponentTypeIdent = ident .
```

4.2 Replications and conditional generation

The replication of a group of statements can be expressed conveniently by the for statement. The identifier following the symbol FOR is a fresh identifier valid only within the replication. It can be used in place of a constant expression.

Example

```
TYPE bo(n) = ARRAY [1..n] OF boolean;
SIGNAL a: COMPONENT (in,out:bo(4)); b:bo(4);

 FOR i:=1 TO 4 DO a.in[i] := b[i] END;
```

This statement can be abbreviated to a.in := b. It is allowed to have parameterized types in Zeus. The actual parameters are numerical constant expressions. Parameterized types are very powerful in connection with recursion see below).

The replication statement is very convenient and is used in most hardware description languages. However it is appropriate to consider the replication statement as part of a meta language which is used to generate hardware. (The analogy to an assembly language with conditional assembly suggests itself.) In the extreme case the meta language is a general purpose programming language which is used to "compute" hardware.

Conditional hardware generation with simple conditions occurs often and therefore we introduce it in Zeus. Conditional hardware generation should also be considered as a part of the meta language. Conditional hardware generation is expressed with WHEN-THEN-OTHERWISE keywords to distinguish it from the conditional statement (to be defined later).

Example

```
FOR i:=2 TO 2*n-1 DO
 WHEN i MOD 2 <> 0 THEN
  se[i](*,(se[i DIV 2].in.contents[1],se[i DIV 2].in.sendup))
 OTHERWISE
  se[i](*,(se[i DIV 2].in.contents[2],*))
```

```
END
```

This is a wiring statement for a binary tree.

The condition after WHEN is a constant expression which is evaluated at compile time. We have essentially adopted the Modula-2 syntax for these constant expressions.

Conditional hardware generation and the parameterized types turn Zeus into a recursive hardware description language. As an example we give the Zeus specification of the routing network described in [Lim(1982)] with the structural hardware description language HISDL.

```
TYPE
 bit(n)=ARRAY[0..10] OF boolean; channel(n)=ARRAY[0..n] OF bit(10);

 router=
 COMPONENT(IN inport0,inport1:bit(10);
       OUT outport0,outport1:bit(10)) IS
 BEGIN ...
 END;

 routingnetwork(n)=
 COMPONENT(IN input: channel(n-1); OUT output: channel(n-1)) IS
 SIGNAL top,bottom: routingnetwork(n DIV 2);
  (* this hardware is only generated
  if it is used in connection or assignment statements later on *)
  c:ARRAY[0..n DIV 2 -1] OF router;
 BEGIN
  WHEN n=2 THEN (* 2*2 router *)
   c[0](input[0],input[1],output[0],output[1])
  OTHERWISE (* decompose routing network into a column of 2*2
   routers and two half-sized sub-networks top and bottom *)
   FOR i:=0 TO n DIV 2 -1 DO
    c[i](input[2*i],input[2*i+1],top.input[i],bottom.input[i]);
    output[i] := top.output[i];
    output[i+ n DIV 2] := bottom.output[i]
   END;
  END;
 END;
```

Syntax

```
replication = FOR ident ":=" ConstExpression
        ( TO | DOWNTO ) ConstExpression DO [ SEQUENTIALLY ]
            StatementSequence END .
condGeneration = WHEN ConstExpression THEN StatementSequence
    { OTHERWISEWHEN ConstExpression THEN StatementSequence }
```

[OTHERWISE StatementSequence] END .

4.3 Connections

A signal s which is an instantiated component type can be "called" in a connection statement. This connection statement establishes (additional) connections between the pins of s (represented by the parameters) and other signals. There may be at most one connection statement for an instantiated component type.

The correspondence between the actual parameters and the formal parameters in the heading of the referenced component is given by the parameters' position. If the formal parameter is specified as an OUT or INOUT parameter, the corresponding actual parameter must be a signal expression,i.e. an S-expression (in the Lisp sense) of signals. If the parameter is specified as an IN parameter, the corresponding actual parameter is an expression, and no assignment to the IN parameter occurs within the component type definition. The formal/actual correspondence for IN and OUT parameters is subject to the rules of assignment with the ":=" operator. If with the given connection statement no additional connections are made for a pin, then a "*" is used as actual parameter.

Connection statements can be translated into assignment statements according to the following rule. Let C be an instantiated component type with basic formal parameters a1,a2, ... ,an. Let C(x1,x2, ... ,xn) be a connection statement. If ai is an OUT parameter (hence of type boolean) then we generate xi:=ai. If ai is an IN parameter (hence of type boolean) then we generate ai:=xi. If ai is an INOUT parameter (hence of type multiplex), we generate ai==xi. It is allowed to specify connections several times as long as they are identical.

The formal/actual correspondence for INOUT parameters is subject to the rules of assignment with the "==" operator. An actual parameter is connected to a formal INOUT parameter by aliasing. An aliasing operation

FormalParameter == ActualParameter

causes a connection to be made between the formal and actual parameter. The consequence is that we have one signal with two (or more) names. Only signals whose basic substructures are of identical type may be aliased. A connection statement which connects a formal INOUT parameter to an actual parameter must not occur within an if statement. (Aliasing cannot be done conditionally.)

Example

```
TYPE
 bo(n) = ARRAY [1..n] OF boolean;
 RandomAccessMemory =
 COMPONENT (IN A:bo(3); OUT DA:bo(9)) IS
```

```
 BEGIN
  ...
 END;
SIGNAL
 RAM:RandomAccessMemory;
 F:bo(9);
BEGIN
 ...
 RAM(*,F); <* connects the output DA of RAM with signal F;
        an equivalent statement is  F:=RAM.DA *>
```

According to the syntax definition, any signal can occur at the beginning of a connection statement. We have to restrict this rule as follows: At the beginning of a connection statement we allow only signals which are the instantiation of a component which has a body (i.e. a non-empty statement part) or signals which are a sequence of instantiated equal components with a body. In the latter case the signal type must be an array of components. If the sequence of instantiated components contains q equal components of type

COMPONENT(x1:t1; ... ;xn:tn)

then in the connection statement the i-th parameter has to contain q times as many basic signals as type ti (1<=i<=n).

Example

```
TYPE
 r=COMPONENT(IN a:boolean; OUT b:boolean) IS
BEGIN
...
END;

SIGNAL
 x:ARRAY[1..10] OF r;
 s,t:ARRAY[1..10] OF boolean;

BEGIN
 x(s,t)  <* or x[1..10](s,t) *>
END
```

This statement part is an abbreviation for

```
BEGIN
 FOR i:=1 TO 10 DO
  x[i](s[i],t[i])
 END
END
```

Syntax

connection = signal [expression] .

4.4 Conditional statement

A hardware description language which is aimed at describing VLSI algorithms (VLSI circuits) should offer a construction for a switch. One possibility is to offer a standard function SWITCH(b,a) which returns the value of a if b is 1 and which has value NOINFL otherwise.

Such a standard function has two disadvantages. First it does not provide an "else" or "otherwise" facility. Second it only allows to formulate conditional assignments but not conditional connections and replications.

Therefore we provide an IF statement with Modula-2 syntax instead of the SWITCH function. The statement

IF b THEN h:=a END

has the same meaning as the assignment

h:=SWITCH(b,a).

All conditions in an IF statement are evaluated in parallel. Although switches are bidirectional devices in many implementations, we allow to use them only in one direction. For example, the statement

F b THEN x==y END

s illegal. This is not a limitation at the level of Zeus programs. Bidirectional witches might be hidden in predefined component types.

Syntax

conditional = IF expression THEN StatementSequence
{ ELSIF expression THEN StatementSequence }
[ELSE StatementSequence] END .

4.5 Sequential and parallel statement

So far the statement order was of no importance; all statements are thought to be executed in parallel. The simulator has the job of finding the proper sequence of execution by generalized topological sorting. However sometimes the user knows that certain statements have to be executed sequentially.

The sequential and parallel statement permit the user to put more useful redundancy into the Zeus text; however they have no semantical implications. The simulator will check whether the specified sequence is compatible with the sequence produced by the semantical definition. If the user knows that a statement sequence S1; S2; ... ;Sn is executed sequentially by the hardware he can specify this with

```
SEQUENTIAL
 S1; S2; ... ;Sn
END
```

The statements Si might contain statements which are executed in parallel. In other words the sequentiality of S1; S2; ... ;Sn is not inherited by the statements nested within the n statements.

The parallel statement is introduced for reversing the effect of the sequential statement. For example if S1 and S2 have to be executed in parallel and afterwards S3; ... ;Sn sequentially the user specifies this with

```
SEQUENTIAL
 PARALLEL
  S1;S2
 END;
 S3; ... ;SN
END
```

A parallel statement which is not nested in a sequential statement has no effect since parallelism is the default.

A special rule applies for the replication statement. If the sequence of statements specified by a replication statement are executed sequentially then the reserved word SEQUENTIALLY is written after DO.

Example

```
SEQUENTIAL
 S1;
 FOR i:=2 TO n DO SEQUENTIALLY S[i] END;
END
```

is equivalent to

```
SEQUENTIAL
 S1; S2; ... ;Sn
END
```

Syntax

sequential = SEQUENTIAL StatementSequence END .
parallel = PARALLEL StatementSequence END .

4.6 With statement

The with statement specifies a signal variable and a statement sequence. In these

statements the qualification of component pin identifiers may be omitted, if they are to refer to the variable specified in the with clause. The with statement opens a new scope as in Modula-2.

The signal which occurs after WITH has to be specified completely. The abbreviation rules which hold for signals in general do not hold here.

Syntax

with = WITH signal DO StatementSequence END .

4.7 Summary of static type rules

In this section we systematically summarize the static type rules which have been scattered through earlier sections. The main purpose of these type rules is to prevent hardware designs in which a direct connection from power to ground could occur. The static type rules have to be supported by run-time checks for signals of type multiplex (and boolean signals which are conditionally assigned several times or which are assigned with "==" and which are either formal OUT parameters or IN parameters of an instantiated component). It is easy to show that deciding whether a signal of type multiplex is assigned the value 0 or 1 exactly once is NP-complete. This is a theoretical justification for the run-time checks.

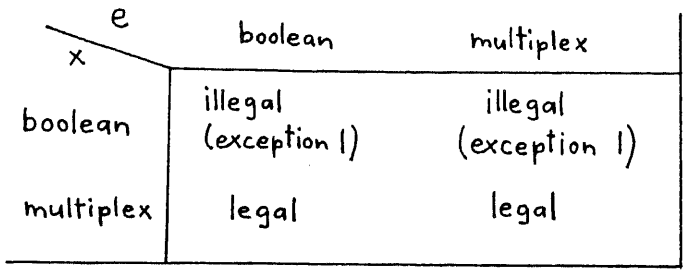
Our static type rules can essentially be reduced to assignment statement rules. Therefore we start with these rules. Let z:=e be an assignment statement, where z is a signal and e an expression. We require that the types of z and e have the same number of basic components. The basic components have a natural order and this specifies the pairs of basic subsignals which are assigned. Therefore it is sufficient to discuss the case x:=e where x is a signal of basic type. x and e can be of type multiplex or boolean.

Unconditional assignment (x:=e)

All four combinations of boolean and multiplex are legal, however there may be no other assignments to x (also not "hidden" assignments within a connection statement). This rule is adopted to prevent direct power ground connections, e.g. x:=1; x:=0.

It should be noted that an unconditional assignment multiplex:= boolean/multiplex abuses the type multiplex. Namely the assignee of type multiplex on the lefthandside cannot obtain any further assignments. However there are situations where this facility is important, e.g. if the first element a[1] of "a:ARRAY[1..n] OF multiplex is initialized by some boolean value, but a[2], ... ,a[n] are assigned conditionally.

Conditional assignment (IF b THEN x:=e END)



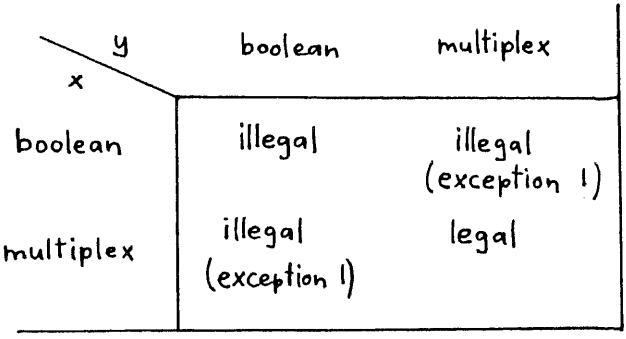
type rules (1)

Exception 1: The boolean signal is a formal OUT parameter or an IN parameter of an instantiated component.

Motivation:

Actually the conditional assignment boolean:=boolean/multiplex should be illegal, however this would make many Zeus programs considerably longer. Assume we have a formal OUT parameter x which is by definition of type boolean. Internally we would like to define it through several conditional assignments. Since it would be cumbersome to define an auxiliary multiplex signal h and assign h to x, we adopt exception 1.

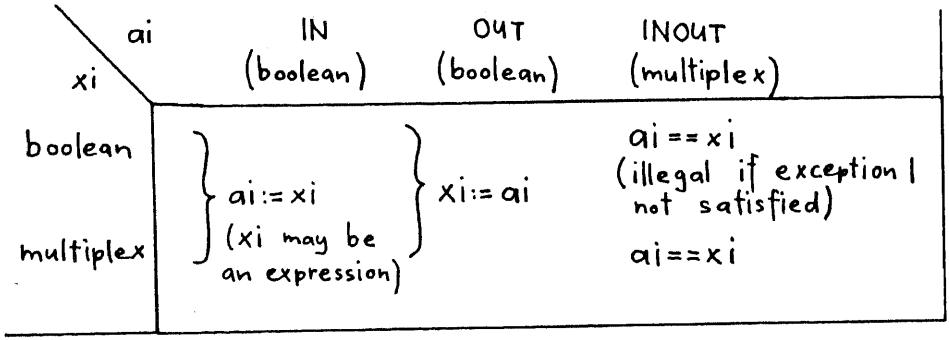
Equivalence of signals (x==y, x and y basic signals)



type rules (2)

Exception 1 is defined and motivated as above. The assignment boolean==boolean is illegal since it would allow direct power ground connections, e.g. a:=1; b:=0; a==b.

It remains to summarize how connection statements and function component calls are reduced to assignment statements. Let f be a (function) component with formal parameters a1,a2, ... ,an. Let f(x1,x2, ... ,xn) be a connection statement for component f or a call of f. If ai is an IN parameter then xi is an expression. If ai is an OUT or INOUT parameter then xi has to be a signal expression, i.e. an S-expression (in the Lisp sense) of signals. The following table specifies the corresponding assignments and the type rule restrictions for assignments of basic signals.



type rules (3)

The types of ai and xi have the same number of basic subtypes, but ai and xi might have different types. Parentheses might have to be used to structure the actual parameters properly. Assume we have a component with n formal

parameters. Then in a connection statement or function call we need n signal expressions (those corresponding to IN parameters might be general expressions) which are separated by n-1 commas. However the parenthesis structure within the n signal expressions is unimportant.

Examples

```
TYPE
h=COMPONENT(IN a:ARRAY[1..5] OF boolean;
        OUT b:COMPONENT(b1,c1,d1,e1,f1: boolean));
SIGNAL s:h;
```

Correct connection statements are (p is a boolean array of 2 elements and q is a boolean array of 3 elements):

```
s((p,q),(p[1],q[2],p[2],q[1],q[3]))
s((p,(1,1,1)),(1,0,1,0,1))
```

5. Storage elements

Storage facilities are introduced in the form of a standard component type.

5.1 Synchronized systems

Storage elements inherently introduce the concept of time. Time is assumed to proceed in discrete steps, so-called clock cycles. In each cycle, all signal values are re-evaluated. Storage elements make it possible to refer to the past, specifically to the previous clock cycle. In Zeus the clock is (essentially) an implicit object and is the same for all storage elements. A system with one clock is called a synchronized system.

A register is a binary storage element with an input "in" and output "out". Its heading is declared implicitely as

```
COMPONENT REG(IN in: boolean; OUT out:boolean);
```

"out" is defined to be equal to the value of parameter "in" in the preceding clock cycle. If "in" is not changed during a clock cycle, it keeps its value.

It is allowed that in the same clock cycle the "in" port is assigned a value and that the stored value (from the last clock cycle) is read at the "out" port.

The clock signal is denoted by the predefined identifier CLK. A signal of type REG may e.g. be implemented by a flip-flop or a dynamic storage element.

The REG component type can be used to describe a random access memory. Let

```
ram: ARRAY[0..1023] OF ARRAY[1..16] OF REG;
```

be a memory with n 16 bit words. Assume that a is a 10 bit address,i.e.

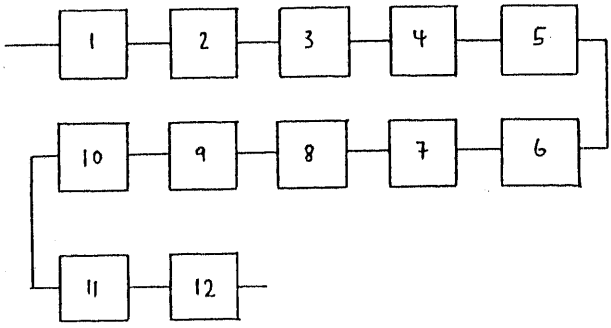


Fig. Snake

6.4 Replacement

For many hardware arrangements it is convenient to think of them in terms of an array of different elements, i.e. a chessboard like configuaration. To make the definition of such hardware easy we introduce a signal type virtual. A signal of type virtual is replaced in a replacement statement (a part of the basic statement) by a real signal, e.g. s=type, where s is a signal of type virtual and "type" is a Zeus type. A signal of type virtual may be replaced at most once.

The replacement statement is not a layout statement; however the layout language is the only proper place for replacements. Perhaps the layout language should be renamed to "layout and replacement" language.

Example

```
TYPE
 black=COMPONENT(IN top,left:boolean;
          OUT bottom,right:boolean) IS ... ;
 white=COMPONENT(IN top,left:boolean;
          OUT bottom,right:boolean) IS ... ;

 chessboard(n)=
 COMPONENT(...) IS
 SIGNAL m: ARRAY[1..n,1..n] OF virtual;
 { ORDER toptobottom
   FOR i:=1 TO n DO
    ORDER lefttoright
     FOR j:=1 TO n DO
      WHEN odd(i+j) THEN
       m[i,j]=black
      OTHERWISE
       m[i,j]=white
      END;
     END;
```

```
    END;
  END;
 END;
}
BEGIN
 ...
 FOR i:=2 TO n-1 DO
  FOR j:=2 TO n-1 DO
   m[i,j](m[i-1,j].bottom,m[i,j-1].right,
        m[i+1,j].top,m[i,j+1].left);
  END;
 END;
 ...
END;
```

7. Zeus syntax

We use the extended Backus-Naur formalism proposed by N. Wirth (see e.g. [Wirth(1982)]).

```
 1 Hardware = {declaration} .
 2 declaration = constDeclaration | typeDeclaration | signalDeclaration .
 3  constDeclaration = CONST { ident "=" constant ";" } .
 4   ident = letter { letter | digit } .
 5   constant =ConstExpression | sigConstExpression.
 6    ConstExpression = SimpleConstExpr [ relation SimpleConstExpr ] .
 7     relation = "=" | "<>" | "<" | "<=" | ">" | ">=" .
 8     SimpleConstExpr = "=" ["+"|"-"] ConstTerm
 9      { AddOperator ConstTerm } .
10     AddOperator = "+" | "-" | OR .
11     ConstTerm = ConstFactor { MulOperator ConstFactor } .
12     MulOperator = "*" | DIV | MOD | AND .
13     ConstFactor = number | "(" ConstExpression ")" | NOT ConstFactor |
14     ident [ "(" ConstExpression { ";" ConstExpression } ")" ] .
15     number = digit { digit } ["B" | "b"] .
16    sigConstExpression = "(" sigConstExpression
17     { "," sigConstExpression } ")" |
18    value | BIN "(" ConstExpression "," ConstExpression ")".
19      value = "0" | "1" | ident .
20 typeDeclaration = TYPE { ident [ "(" idlist ")" ] "=" type ";" }.
21  type = arrayDeclaration | componentDeclaration | ident
22       ["("ConstExpressionList")"].
```

```
23    arrayDeclaration =
24      ARRAY "[" ConstExpression ".." ConstExpression "]"  OF type .
25    componentDeclaration =
26      COMPONENT "(" [ fparams { ";" fparams } ] ")"
27        [ "{" layoutStatementList "}" ]
28        [ [ ":" type  ] IS [ USES idlist ";" ] { declaration }
29        [ "{" layoutStatementList "}" ] BEGIN StatementSequence END ] .
30      fparams = [ IN | OUT ] fieldlist .
31        fieldlist = idlist ":" type .
32        idlist = ident { "," ident } .
33      StatementSequence =  statement { ";" statement } .
34        statement = [ assignment | replication | condGeneration |
            connection |
35            conditional | result | parallel | sequential | with ] .  /empty/
36          assignment = signal ( ":=" | "==" ) expression .
37            signal = ( ident { "[" (ConstExpression
38              [ ".." ConstExpression ] "]"|
39              NUM "(" signal ")" ) "]" | "." ident [ ".." ident ] } )
              | "*" .
40            expression = signal | functionComponentTypeIdent
41              ["("ConstExpressionList")"]
42                [ expression ] | BIN "(" ConstExpression ","
43                ConstExpression")" |
44                sigConstExpression | "*" [ ":" ConstExpression ] |
45                "(" expression { "," expression } ")" .
46              functionComponentTypeIdent = ident .
47          replication = FOR ident ":=" ConstExpression
48                  ( TO | DOWNTO ) ConstExpression DO
49                    [SEQUENTIALLY] StatementSequence END .
50          condGeneration = WHEN ConstExpression THEN
                StatementSequence
51              { OTHERWISEWHEN ConstExpression THEN
                StatementSequence }
52              [ OTHERWISE StatementSequence ] END .
53          connection = signal [ expression ] .
54          conditional = IF expression THEN StatementSequence
55                    { ELSIF expression THEN StatementSequence }
56                    [ ELSE StatementSequence ] END .
57          result = RESULT expression .
58          parallel = PARALLEL StatementSequence END .
59          sequential = SEQUENTIAL StatementSequence END .
60          with = WITH signal DO StatementSequence END .
61    ConstExpressionList = ConstExpression { "," ConstExpression } .
62  signalDeclaration =
63    SIGNAL { idlist ":" type [ "(" ConstExpressionList ")" ] ";" } .
```

"LayoutStatementList" is defined in the layout language syntax (see below). "digit" and "letter" have the standard interpretation. In the following cross-reference listing the negative line numbers indicate where a symbol has been defined.

```
arrayDeclaration          -23   21
assignment                -36   34
AddOperator               -10    9
componentDeclaration      -25   21
condGeneration            -50   34
conditional               -54   35
connection                -53   34
ConstExpression            61   61    51    50    48    47    44    43
                           42   38    37    24    24    18    18    14
                           14   13    -6     5
ConstExpressionList        63  -61    41    22
ConstFactor                13  -13    11    11
ConstTerm                 -11    9     8
constant                   -5    3
constDeclaration           -3    2
declaration                28   -2     1
digit                      15   15     4    (standard definition)
expression                 57   55    54    53    45    45    42   -40
                           36
fieldlist                 -31   30
fparams                   -30   26    26
functionComponentTypeIdent -46  40
Hardware                   -1
ident                      47   46    39    39    37    32    32    21
                           20   19    14    -4     3
idlist                     63  -32    31    28    20
layoutStatementList        29   27          (see layout language syntax)
letter                      4    4          (standard definition)
MulOperator               -12   11
number                    -15   13
parallel                  -58   35
relation                   -7    6
replication               -47   34
result                    -57   35
sequential                -59   35
SimpleConstExpr            -8    6     6
StatementSequence          60   59    58    56    55    54    52    51
                           50   49   -33    29
sigConstExpression         44   17    16   -16     5
signal                     60   53    40    39   -37    36
signalDeclaration         -62    2
statement                 -34   33    33
type                       63   31    28    24   -21    20
typeDeclaration           -20    2
value                     -19   18
with                      -60   35
```

Predefined function component types
AND,NAND,OR,NOR,NOT,XOR,EQUAL
RANDOM (for describing bistable elements)

Predefined component types
REG

Predefined signals
CLK, RSET
Predefined functions for constant expressions
min, max, odd

Layout language syntax

```
 1 layoutStatementList = layoutStatement { ";" layoutStatement } .
 2  layoutStatement = [ basic | order | replication | boundary |
      condGeneration | with ].
 3   basic = [ orientationchange ] signal = type .
 4    orientationchange = ident .
 5   order = ORDER directionOfSeparation layoutStatementList END .
 6    directionOfSeparation = ident .
 7   replication = FOR ident ":=" numConstExpression
     (TO | DOWNTO) numConstExpression DO
 8           layoutStatementList END .
 9   boundary = TOP | RIGHT | BOTTOM | LEFT layoutStatementList .
10   condGeneration = WHEN ConstExpression THEN layoutStatementList
11    { OTHERWISEWHEN ConstExpression THEN layoutStatementList }
12    [ OTHERWISE layoutStatementList ] END .
13   with = WITH signal DO layoutStatementList END .
```

basic	-3	2						
boundary	-9	2						
ConstExpression	11	10		(see main syntax)				
condGeneration	-10	2						
directionOfSeparation	-6	5						
ident	7	6	4	(see main syntax)				
layoutStatement	-2	1	1					
layoutStatementList	13	12	11	10	9	8	5	-1
numConstExpression	7	7		(see main syntax)				
order	-5	2						
orientationchange	-4	3						
replication	-7	2						
signal	13	3		(see main syntax)				
type	3			(see main syntax)				
with	-13	2						

Directions of separation
toptobottom, bottomtotop, lefttoright, righttoleft, toplefttobottomright, bottomrighttotopleft, toprighttobottomleft, bottomlefttotopright.

Orientation changes (counter clock wise)
All operations of the dihedral group, except the identity: rotate90, rotate180, rotate270, flip0, flip45, flip90, flip135.

8. Semantics

Several formal mechanisms have been proposed to describe the semantics of Zeus-like languages. The semantics could be described by a restricted form of Petri-nets as in [Misunas(1973)] or by data-flow schemas (elementary computation schemas) [Dennis(1974)]. An elementary computation scheme is essentially an abstraction of a Zeus component. We prefer the use of an informal semantical description. First we define the semantics of the functional part of Zeus and later the semantics of the layout part.

We introduce the functional definition of Zeus with a comparison of CS networks [Hayes(82)] and Zeus components. Zeus components correspond essentially to restricted CS (connector-switch) networks. Only a subset of the CS networks can be expressed in Zeus since at most one (0,1,UNDEF)-assignment is allowed to a signal of type multiplex. We believe that this is a reasonable restriction which makes the design of hardware safer.

A positive switch in CS theory is represented by the if statement

IF k THEN d1:=d2

and a negative switch by

IF NOT k THEN d1:=d2.

Connectors in CS theory are represented by connection and assignment statements in Zeus.

For example, a regular nor-gate cannot be expressed in terms of switches and connections in Zeus (however note that the nor-function is predefined). The reason is that we would have a sequence

IF a THEN x:=0 END; IF b THEN x:=0 END

which is illegal if both a and b are one.

We define the semantics of Zeus by describing a simulator for Zeus programs. The basic problem of defining a sequential simulator is that hardware is inherently parallel. We perform a straight-forward simulation without delay information.

It is sufficient to describe the simulation of one component. First we translate a Zeus component type definition into a node-labeled directed graph. The nodes of this graph represent signals, the predefined components and if statements. The directed edges indicate how the signals are evaluated in a generalized topological order (in a wide sense).

A Zeus component C is translated into a directed semantics graph G(C) which contains one node for each basic signal local to C, one node for each predefined component (AND, ... , REG) and one node for each if statement. The local signals include: 1. The pins of C, 2. the pins of components local to C and 3. the other signals which are declared local to C.

The predefined component REG has a special interpretation: It has neither internal nodes nor edges and acts in this way as a cycle breaker.

The assignment statements with the operator ":=", function component instances and connection statements introduce directed edges.

An assignment statement of the form s1:=s2, where s1 and s2 are signals of basic type introduces a directed edge s2 -> s1. This rule generalizes to structured signals. An expression e of the form "ident(a,b)" is evaluated in the following way (a,b signals of basic type):

If ident is the name of a predefined component we introduce a node with the name of the component in the semantics graph. This node has two entering edges ınd one edge which exits.

f ident=AND then the exiting edge carries a 0 as soon as one entering edge is 0. The exiting edge carries a 1 iff both entering edges are 1. In all other cases the output is UNDEF.

If ident=NAND then the exiting edge carries a 1 as soon as one entering edge is 0. The exiting edge carries a 0 iff both entering edges are 1. In all other cases the output is UNDEF.

If ident=OR then the exiting edge carries a 1 as soon as one entering edge is 1. The exiting edge carries a 0 iff both entering edges are 0. In all other cases the output is UNDEF.

A similar rule holds for ident=NOR.

If ident=XOR a and b have to be defined (0 or 1) to get output 0 or 1. The exiting edge has the value 1 iff a and b are different. In all other cases the output is UNDEF.

If ident=EQUAL also both a and b have to be defined (0 or 1) to get output 0 or 1. The exiting edge is 1 iff a and b are equal. In all other cases the output is UNDEF.

For the NOT component the obvious rule holds.

These rules are easily generalized to structured signals and to components which

have more than two parameters (the above rules hold for each substructure of basic type).

User defined function components are translated into a semantic graph and evaluated by applying the above rules recursively.

An assignment statement of the form a1 := f (b1,b2, ... ,bn) where a1 is a signal of basic type, introduces a semantics graph for the function component f and the exiting edge of this graph is directed towards a1. This rule generalizes to assignments involving structured signal variables (remove the structure and consider the natural sequence of basic signals). The RESULT statement has a similar interpretation as the assignment statement. The assignment statement x==y identifies the two signals x and y.

In order to define the semantics of the IF-THEN-ELSE statement we reformulate it as a sequence of IF-statements.

The Zeus statement

IF b1 THEN s1
ELSIF b2 THEN s2
...
ELSIF bn-1 THEN sn-1
ELSE sn
END

is equivalent to

IF b1 THEN s1 END;
IF AND(NOT b1,b2) THEN s2 END;
...
IF AND(... AND(NOT b1,NOT b2), ... ,bn-1) THEN sn-1 END;
IF AND(... AND(NOT b1,NOT b2), ... ,NOT bn-1) THEN sn END;

To describe the semantics of

IF b1 THEN s1 END

we distinguish three cases:

a) assignment: If s1 is an assignment s:=e then we need to know the values of b1 and perhaps of e before we can make the assignment. Therefore we introduce an if node in the semantics graph which has two entering edges, one coming from b1 the other from e. The exiting edge goes to signal s. As soon as b1 is 0, signal s has value NOINFL (see below). IF b1 is 1 then e has to be evaluated and its value is transmitted to s.

b) connection: If s1 is a connection statement it can easily be rewritten as a sequence of assignment statements for which rule a) applies.

c) replication: If s1 is a replication statement we apply rules a) and b) to the

connection and assignment statements within the replication statement.

Connection statements introduce arrows according to the following rule. Let C be a component with basic formal parameters a1,a2, ... ,an. Let C(x1,x2, ... ,xn) be a connection statement. Then there is an arrow from ai to xi or vice versa according to the IN/OUT modes of ai: OUT -> , IN <- . In other words, if ai is an OUT parameter then there is an arrow ai->xi. If ai is an IN parameter then there is an arrow xi->ai.

INOUT parameters don't introduce arrows but they identify signals. If one signal in a class of identified signals is assigned a value then all signals in the class are assigned the same value. Function component instances introduce arrows in a similar way as connection statements. We require that the resulting graph G(C) defines a partial order on the nodes.

A signal can have four values in our model of computation: 0, 1, UNDEF (undefined), NOINFL (no influence). Only a signal of type multiplex can have value NOINFL. We define how these values behave for the predefined function components AND, NAND, OR, NOR, NOT, XOR, EQUAL: If these predefined components don't have a 0 or 1 value according to rules defined earlier then the result has value UNDEF. In the IF statement

IF b THEN s:=e END

the value of s is NOINFL if b=0, otherwise if b=1 the value of s is the value of e. If b=NOINFL then s has value UNDEF. This rule generalizes in the obvious way to other IF statements since they can be rewritten as IF statements of the above form.

Finally we have to define how signals behave under conditional simultaneous assignments. Let x be a basic signal variable which is assigned a value several times. Value NOINFL is overruled by any other value. If UNDEF is assigned to x then x has value UNDEF independent of other assignments. If x is assigned several times 0,1 or UNDEF at runtime then x has value UNDEF and an error message is given.

For evaluating the signals we cannot rely on a regular topological order since the signals determine the evaluation sequence. The evaluation starts with all the nodes (signals) which have no predecessor (signals corresponding to input pins, the out pin of registers or signals defined by constant signal expressions). These signal values are propagated according to the rules specified above for the predefined components and the if nodes.

The signal propagation is best described in terms of firing rules. A node of the semantics graph is firing under the following conditions: If it is a

a) signal node: If the signal node is of type boolean then it is firing on the exiting edges as soon as it is assigned a value (0,1,UNDEF). If the signal node is of type multiplex then it is firing as soon as all incoming edges have been assigned (0,1,UNDEF,NOINFL). The "strongest" signal survives as described earlier.

b) predefined function component node: The node is firing on its exiting edge as soon as the value of the function is determined according to the rules described earlier. For example the node corresponding to AND(b,c) is firing on its exiting edge as soon as b or c is assigned 0; it is firing 1 if both b and c are assigned 1; otherwise it is firing UNDEF after b and c have been assigned.

c) IF-node: The node is firing as soon as both entering edges have been assigned.

There are many ways of propagating the signals sequentially; however all will lead to the same result.

Note that a subcomponent may be executed before its super component has all inputs assigned. Therefore this semantical definition sort of ignores the hierarchical structure of components. However it is possible to implement the simulator in such a way that it obeys the hierarchical structure and still gives the same result. Each component type is considered to be a package (in the Ada sense) which exports a procedure (representing the functionality) and which stores the register values internally. A signal declaration corresponds to an instantiation of the (generic) package. A call of the exported procedure will be performed after all the IN and INOUT parameters have been assigned.

Example

```
TYPE
c=
COMPONENT(IN a,b,c,x,y,rin: boolean;
      OUT rout: boolean; out: multiplex) IS
SIGNAL r:REG;
BEGIN
 IF x THEN out:=AND(a,b);
 IF y THEN out:=c;
 r(rin,rout)
END;
```

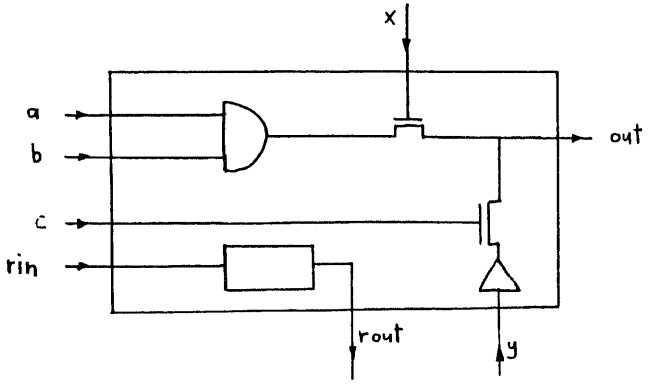


Fig. c

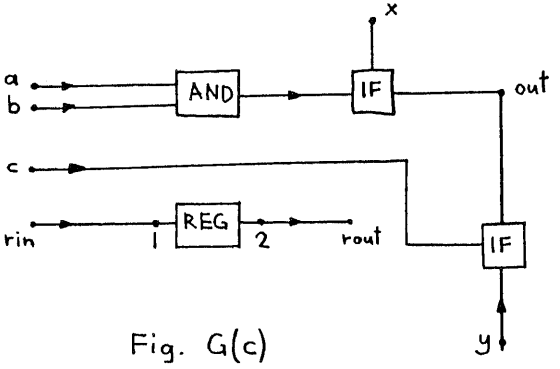


Fig. G(c)

A possible evaluation sequence is

2(0),rout(0),rin(1),1(1),a(1),c(0),b(1),x(1),y(1),out(1)

Now we sketch the semantics of the layout language. We use the abbreviation "x1 is left of x2" for "the right edge of the bounding rectangle of x1 is left of the left edge of the bounding rectangle of x2". Each order, and replication statement defines a bounding rectangle. This bounding rectangle contains all the components referred to in the order or replication statement. To explain the semantics of the order statement

```
ORDER lefttoright
 x1;x2
END
```

recursively we distinguish the following cases:

1. x1 and x2 are both components

Component x1 is left of component x2.

2. x1 is a component and x2 a replication statement FOR i:=1 TO n DO y[i] END

Component x1 is left of element y[1], where y[1] is the first element of the replication statement. Element y[i] is left of y[i+1], i=1, ... ,n-1.

3. x1 is a component and x2 is an order statement

x1 is left of x2.

Other combinations and the other directions of separation (besides lefttoright) have similar semantics.

9. Relation to previous work

A large number of hardware description and layout languages have been proposed and it is impossible to compare Zeus to all of them. Some comparisons have already been given in the introduction. We believe that Zeus is unique in the sense that it incorporates many of the lessons which have been learned in the design and usage of programming languages like Pascal and Modula-2 [Wirth(1982a)]. Zeus offers only a few concepts: constants (signal and numerical), types (components, arrays, multiplex, boolean), signals, statements (assignment, connection, conditional, replication, conditional generation) which are sufficient for describing "reasonable" synchronized circuits at a level which is close to the actual hardware.

For some applications Zeus is either too high-level or too low-level. OCCAM ([INMOS(1982a)], based on Hoare's communicating sequential processes) is a language which is at a higher level than Zeus. OCCAM is a simple programming language, based on the concepts of concurrency and communication, providing a close relationship between a program and its implementation. OCCAM is an interesting front-end for Zeus.

For applications where the functional part of Zeus is too high-level (since it is technology independent) Zeus is easily extended by a few basic types and corresponding assignment rules. We propose to extend Zeus in such a way that CSA networks [Hayes(1982)] can be expressed. With such an extension Zeus is suitable for describing MOS circuits down to the transistor level.

One might go further and add layout language features to Zeus which are available in layout languages like MULGA [Weste (1981)], ALI [Lipton(82)], HILL [Lengauer(1982)] and others.

Zeus was developed with the following applications in mind:

1. Communication tool

Many VLSI circuits have been described informally in the literature but only a few of them have been written down in a formal language like Zeus. One advantage of a formal, easy to read, and technology relevant notation is that the circuits are studied more thorougly. Different solutions can be analyzed and compared and the dissemination of circuits is facilitated. The reader is invited to use Zeus to describe for example the circuits which have been published in the following papers: [Ahmed (1982), Cappello(1982), Cappello/Steiglitz(1981), Floyd/Ullman (1982), Foster/Kung(1981), Fuchs(1982), Guibas(1979), Guibas(1982), Hambrusch(1981), Hambrusch/Simon (1981), Hwang(1982), Kung(1980), Levitt(1972), Ottman(1982), Rosenfeld(1983), Thompson(1981)].

2. Input language to a simulator

A simulator at the Zeus level is a well understood subject, see e.g. [Breuer/Friedman (76), Nestor/Thomas(1982)]; for interactive simulation see [Sakai(1982)].

3. Input language to an interactive silicon compiler

Developing silicon compilers is a very active research area today. We plan to use work reported e.g. in [Trimberger (1981a,b), Valdes(1982), Rivest(1982)].

Regarding the interaction we plan to use a Lilith-like machine [Wirth(1981)] (language-oriented architecture, high-resolution screen, mouse) and a systematic approach for the design of the interaction like the one used in XS-1 [Nievergelt et al. (1982)]. A syntax directed editor is currently being implemented under XS-1 by Carlo Muller.

Zeus provides the user with the capability to algorithmically define hardware. But a Zeus program, although it contains a complete specification, is not intuitive if it is not accompanied by a picture which represents the components (by rectangles) and the connections (by wires). Interactive graphics systems, on the other hand, allow the user to debug in the form in which he sees the design, but severely restrict the language he may use to express the graphics; he cannot express parameterized components. What is really needed is an interactive system that combines the language and graphics modifications to the data. Such a system has already been investigated but for a language which is at a much lower level than Zeus [Trimberger (1981b)].

10. EXAMPLES

Blackjack, a finite state machine example

```
TYPE
bo5 = ARRAY [1..5] OF boolean;

<* available:
 REG = COMPONENT (IN in:boolean; OUT out:boolean)
  REG.out is REG.in of the previous clock cycle.
 plus, minus = COMPONENT (IN term1, term2:bo5):bo5
 ge,lt = COMPONENT (IN term1, term2:bo5): boolean
*>
blackjack = COMPONENT ( IN ycard:boolean;
        IN value:bo5; OUT hit, broke, stand:boolean) IS
 CONST start=(0,0,0); read=(0,0,1); sum=(0,1,0);
     firstace=(0,1,1); test=(1,0,0);
     end=(1,0,1); zero5 = (0,0,0,0,0); ten = BIN(10,5) ;
 TYPE reg(n) = ARRAY [1..n] OF REG;
 SIGNAL score, card:reg(5); ace:REG; state:reg(3);
      scorelt22, scorege17 : boolean;
 BEGIN
  IF RSET THEN state.in:=start
  ELSE <* the multiplex assignment rule might be
       violated without this else *>
   scorelt22 := lt(score.out,BIN(22,5));
   scorege17 := ge(score.out,BIN(17,5));
   <*state=start*>
   IF EQUAL(state.out,start) THEN
    score.in:=zero5; ace.in:=0; state.in :=read
   END;
   <*state=read*>
   IF EQUAL(state.out,read) THEN
    card.in:=value; hit:=1;
    IF ycard THEN state.in := sum END;
   END;
   <*state=sum*>
   IF EQUAL(state.out,sum) THEN
    score.in:=plus(score.out,card.out);state.in:=firstace
   END;
   <*state=firstace*>
   IF EQUAL(state.out,firstace) THEN
    state.in := test;
    IF AND(EQUAL(card.out,BIN(1,5)),NOT ace.out) THEN
     score.in :=plus(score.out,ten);
     ace.in:=1;
```

```
    END;
   END;
  <*state=test*>
   IF EQUAL(state.out,test) THEN
    IF NOT scoregel7 THEN state.in := read
    ELSIF scorelt22 THEN state.in:=end
    ELSIF ace.out THEN
     <* state.in:=test; *>
     score.in:=minus(score.out,ten);<* ace.in:=1 *>
    END;
   END;
  <*state=end*>
   IF EQUAL(state,end) THEN
    IF scorelt22 THEN stand:=1 ELSE broke:=1 END;
    IF ycrd THEN state.in:=start ELSE state.in:=end END;
   END;
 END
END <* blackjack *>
```

Adders

```
TYPE
 halfadder =
 COMPONENT (IN a,b: boolean; OUT cout,s: boolean) IS
  BEGIN s := XOR(a,b);
    cout := AND(a,b)
  END;

 fulladder =
 COMPONENT (IN a,b,cin: boolean; OUT cout,s: boolean) IS
  SIGNAL h1,h2:halfadder;
  BEGIN h1(a,b,*,h2.a); h2(h1.s,cin,*,s);
   cout := OR(h1.cout,h2.cout)
  END;

 bo(n) = ARRAY [1..n] OF boolean;
 rippleCarry4 =
 COMPONENT (IN a,b:bo(4); IN cin: boolean;
        OUT cout: boolean; OUT s:bo(4)) IS
  SIGNAL add : ARRAY [1..4] OF fulladder; h : bo(5);
  { ORDER lefttoright
    FOR i:=1 TO 4 DO add[i] END
   END
  }
  BEGIN
   SEQUENTIAL
    h[1] := cin;
    FOR i:=1 TO 4 DO SEQUENTIALLY
     add[i] (a[i],b[i],h[i],h[i+1],s[i]);
    END;
    cout := h[5];
   END
  END

is equivalent to (if length = 4)
 rippleCarry (length) <* without auxiliary array h *> =
 COMPONENT (IN a,b:ARRAY[1..length] OF boolean;
        IN cin: boolean; OUT cout: boolean;
        OUT s:ARRAY[1..length] OF boolean) IS
  SIGNAL add : ARRAY [1..length] OF fulladder;
  { ORDER lefttoright
    FOR i:=1 TO length DO add[i] END
   END
  }
  BEGIN
```

```
    SEQUENTIAL
     add[1](a[1],b[1],cin,* <* add[2].cin *>,s[1]);
     FOR i:=2 TO length-1 DO SEQUENTIALLY
      add[i] (a[i],b[i],add[i-1].cout,add[i+1].cin,s[i]);
      <* alternative:
          add[i].a:=a[i]; add[i].b:=b[i]; add[i].cin:=add[i-1].cout;
          add[i+1].cin:=add[i].cout; s[i]:=add[i].s *>
     END;
     add[length](a[length],b[length],
      * <*add[length-1].cout*> ,cout,s[length]);
    END
  END

SIGNAL adder:rippleCarry(4);
```

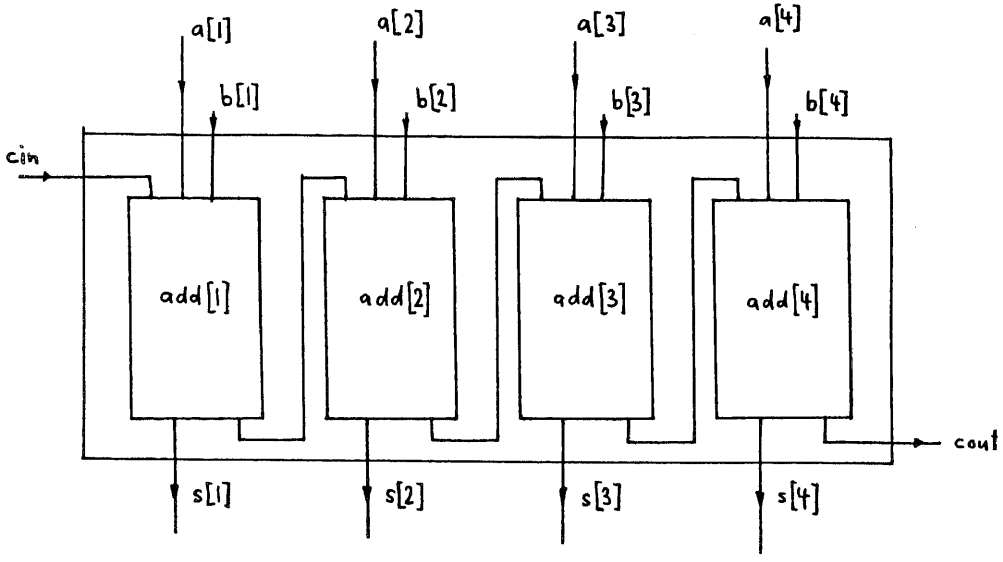


Fig. Adder

Binary Trees

```
TYPE
q =
COMPONENT (IN in: boolean; OUT out1,out2: boolean ) IS
 BEGIN ...
 END;
tree (n) <* n a power of 2 *> =
COMPONENT (IN in: boolean; OUT leaf:ARRAY [1..n] OF boolean) IS
 SIGNAL h: ARRAY [1..n-1] OF q;
 BEGIN
  h[1].in := in;
  FOR i:=1 TO n DIV 2 - 1 DO
   h[i](*,h[2*i].in,h[2*i+1]);
  END;
  FOR i:= 1 TO n DIV 2 DO
   h[i + n DIV 2 -1](*,leaf[2*i-1],leaf[2*i]);
  END;
 END;

SIGNAL a:tree(4);
```

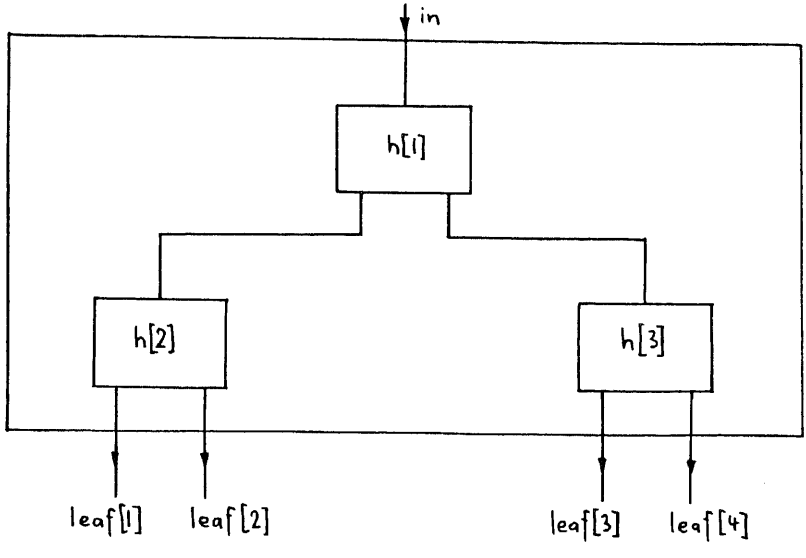


Fig. binary tree
(n=4)

The following is an equivalent recursive definition of a binary tree (but with layout information):

```
TYPE
tree(n) = <* n a power of two, n >= 2 *>
COMPONENT(IN in:boolean; OUT leaf:ARRAY[1..n] OF boolean) IS
SIGNAL
 left, right: tree(n DIV 2);
 preleaf: ARRAY[1 .. n DIV 2] OF q; root:q
 { ORDER toptobottom
    root;
    ORDER lefttoright
     left;right
    END;
  END
 }
BEGIN
 WHEN n>2 THEN
  root.in:=in; left.in:=root.out1; right.in:=root.out2;
  FOR i:= 1 TO n DIV 2 DO
   leaf[2*i-1]:=preleaf[i].out1;
   leaf[2*i]:=preleaf[i].out2;
  END;
  FOR i:=1 TO n DIV 4 DO
   preleaf[i].in:=left.leaf[i];
   preleaf[i+n DIV 4]:=right.leaf[i+n DIV 4]
  END;
 OTHERWISE <* n=2 *>
  root.in:=in; leaf[1]:=root.out1; leaf[2]:=root.out2
 END
END;

SIGNAL a: tree(4);
```

The following component type htree describes the well-known H-tree which has a linear layout area.

```
htree(n)= <* binary tree with n leafs, n a power of 4 *>
COMPONENT(IN in:boolean; out: multiplex) { BOTTOM in;out } IS
```

```
TYPE
 leaftype=
 COMPONENT(IN in:boolean; out: multiplex) { BOTTOM in;out }IS
 BEGIN
  ...
 END;
SIGNAL
 s: ARRAY[1..4] OF htree(n DIV 4); leaf: leaftype;
 { ORDER lefttoright
    ORDER toptobottom
     s[1]; flip90 s[3]
    END;
    ORDER toptobottom
     s[2]; flip90 s[4]
    END;
   END
 }
BEGIN
 WHEN n>1 THEN
  FOR i:=1 TO 4 DO
   s[i].in:= in; out == s[i].out
  END
 OTHERWISE
  leaf.in:= in; out == leaf.out
 END
END;

SIGNAL a: htree(4);
```

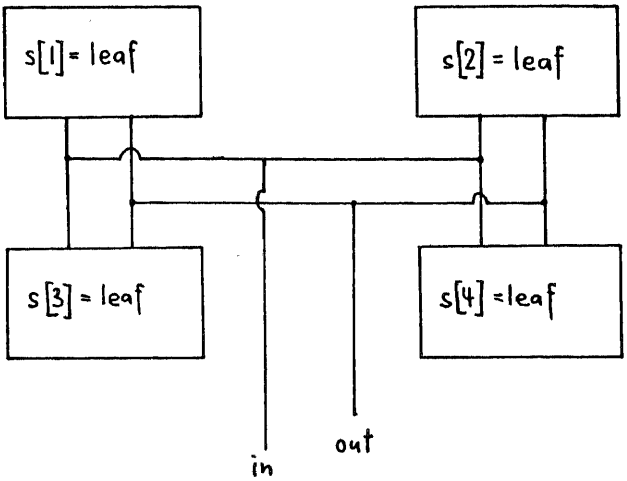


Fig. htree(4)

Pattern Matching

```
<* for 2 letter alphabet only [Foster/Kung (1979)]
It is assumed that string and pattern enter bitwise every second clock cycle.
During an idle input phase we assume that 0's go into the circuit*>

TYPE
patternmatch (length) = <* length odd *>
COMPONENT (IN pattern, string, endofpattern, wild, resultin: boolean;
        OUT result,endout,stringout, wildout, patternout : boolean) IS
TYPE
 comparator =
 COMPONENT (IN pin,sin: boolean; OUT pout, dout, sout: boolean) IS
  SIGNAL p,s: REG;
  BEGIN
     p(pin,pout); s(sin,sout);
     dout := AND(1,EQUAL(p.out,s.out));
     <* the AND could be deleted for the 2 letter alphabet case *>
  END
```

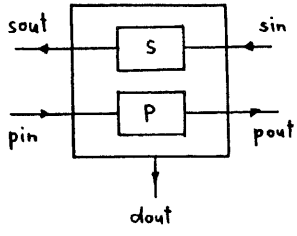


Fig. Comparator

```
 accumulator =
 COMPONENT (IN d,lin,xin,rin: boolean;
         OUT lout,xout,rout:boolean) IS
  SIGNAL tp <* temporary result *>,l,x,r:REG;
  BEGIN
    IF RSET THEN
      tp.in:=1
    ELSE
      l(lin,lout); x(xin,xout); r(rin,*);
```

```
    IF l.out THEN rout:=AND(tp.out,OR(x.out,d)); tp.in:=1
          ELSE rout:=r.out; tp.in:= AND(tp.out,OR(x.out,d))
    END;
  END;
END;
```

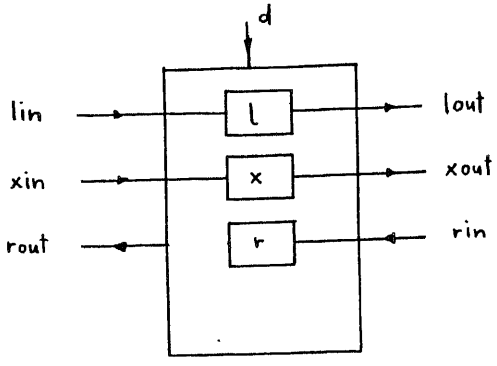


Fig. Accumulator

```
SIGNAL pe:ARRAY[1..length] OF
  COMPONENT(comp: comparator; acc: accumulator) IS
  BEGIN acc.d:=comp.dout
  END;
{ ORDER lefttoright
    FOR i:=1 TO length DO
     ORDER toptobottom
       WITH pe[i] DO
         comp; acc
       END;
     END;
    END
  END
}
BEGIN
  SEQUENTIAL
    (*Connections to outside*)
```

```
    WITH pe[1] DO
      comp.pin := pattern; acc.lin := endofpattern; acc.xin := wild;
      result := acc.rout ; stringout:= comp.sout;
    END;
    WITH pe[length] DO
      patternout:= comp.pout; comp.sin := string ; wildout := comp.xout;
      comp.rin := resultin; endout := acc.lout;
    END;
    resultin:=0;
  END;
  <*Internal connections*>
  FOR i:=2 TO length-1 DO
    WITH pe[i] DO
      comp(pe[i-1].comp.pout,pe[i+1].comp.sout,
          pe[i+1].comp.pin,*,pe[i-1].comp.sin);
      acc(*,pe[i-1].acc.lout,pe[i-1].acc.xout,pe[i+1].acc.rout,
          pe[i+1].acc.lin,pe[i+1].acc.xin,pe[i-1].acc.rin);
    END
  END
END

SIGNAL match: patternmatch(3);
```

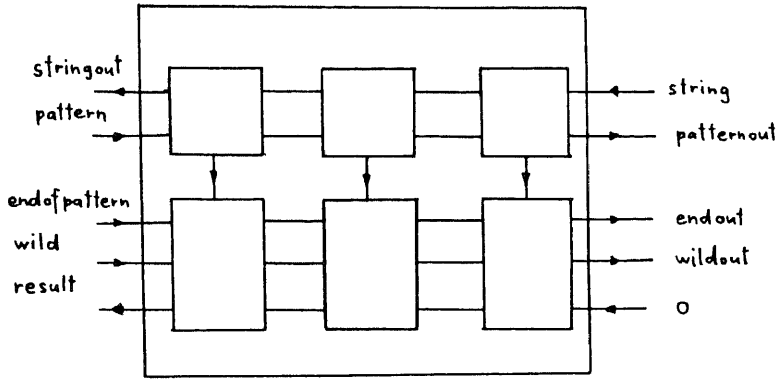


Fig. patternmatch

$^1S_2$	0	S_3
P_3	0	P_2
0	$^2S_3$	0
0	P_3	0
$^2S_3$	0	$^3S_4$
P_1	0	P_3
0	$^3S_4$	0
0	P_1	0
$^3S_4$	0	S_5
P_2	0	P_1

A possible computation sequence. The numbers ^{1,2,3} represent result bits.

Acknowledgements

We would like to thank Niklaus Wirth for encouraging this work. Many thanks to Michael Fitchett, Tarck Ibrahim, Carlo Muller and Peter Schulthess for their comments and suggestions to earlier versions of this paper.

REFERENCES

Ahmed (1982) H.M. Ahmed, "Signal processing algorithms and architectures", Dissertation, Department of Electrical Engineering, Stanford University.

Bode/Händler(80) A. Bode, W. Händler, "Rechnerarchituktur", Springer, 1980.

Breuer(1976) M.A.Breuer, "Digital System Design Automation", Computer Science Press, 1976.

Breuer/Friedman(76) M.A. Breuer, A.D. Friedman, "Diagnosis and reliable design of digital systems", Computer Science Press, 1976.

Bryant(1981) R.E. Bryant, "A switch-level simulation model for integrated logic circuits", VLSI Conference, Edinburgh 1981, pp 329-340.

Cappello(1982) P.R. Cappello, "VLSI architectures for digital signal processing", Dissertation, Department of Electrical Engineering and Computer Science, Princeton University.

Cappello/Steiglitz(1981) P.R Cappello, K. Steiglitz, "Digital Signal Applications of Systolic Algorithms", CMU Conference on VLSI Systems and Computations, pp. 245-254.

Dennis(1974) J.B. Dennis, J.B. Fosseen, J.P. Lindermann, "Data Flow Schemas", in: Proc. of Symposium on theoretical programming, Novosibirsk, Lecture Notes in Computer Science, Vol.19, pp 187-216, Springer Verlag, Berlin 1974.

Duley(1968) J.R. Duley, D.L. Dietmeyer, "A digital system design language (DDL)", IEEE Trans. Comp., Vol. C-17, 1968, pp 850-861.

Floyd(1982) R.W. Floyd, J.D. Ullman, "The compilation of regular expressions into integrated circuits", Journal of the ACM, July 82, Vol. 29, No.3, pp 603-622.

Foster/Kung(79) M.J. Foster,H.T. Kung, "Design of Special-Purpose VLSI Chips: Examples and Opinions", Dep. of Comp. Science, CMU-CS-79-147.

Foster/Kung(81) M.J. Foster, H.T. Kung, "Recognize regular languages with programmable building-blocks", VLSI Conference, Edinburgh 1981.

Foster(1981) M.J. Foster, "Syntax-Directed Verification of Circuit Function", Proc. CMU Conference on VLSI Systems and Computations, Ed. Kung,Sproull, Steele, pp. 196-202, 1981.

Fuchs(1982) H. Fuchs, J. Poulton, A. Paeth, A. Bell, "Developing Pixel-planes, a Smart Memory Based Raster Graphics System, 1982 Conf. on Advanced Research in VLSI, MIT, pp. 137-146.

Guibas(1979) L.J. Guibas, H.T. Kung, C.D. Thompson, "Direct VLSI implementation of combinatorial algorithms", Proc. Caltech Conf. VLSI, 1979, pp 509-525.

Guibas(1982) L.J. Guibas, F.M. Liang, "Systolic Stacks, Queues and Counters", Proc. 1982 Conference on advanced research in VLSI, MIT. pp.155-164.

Hambrusch(81) S.E. Hambrusch, "VLSI algorithms for the connected component problem", Computer Science Department, Penn. State Univ., CS-81-09, March 1981.

Hambrusch/Simon(81) S.E. Hambrusch, J. Simon, "Solving undirected graph problems on VLSI", Computer Science Department, Penn. State Univ., CS-81-23, Sept. 1981.

Hayes(1982) J.P Hayes, "A unified switching theory with applications to VLSI design", Proc. of the IEEE, Vol.70, No. 10, Oct. 1982, pp. 1140-1151.

Hill(1979) D.D.Hill, "ADLIB: A modular, strongly-typed computer design language", Computer Hardware Description Languages (ACM/IEEE) 1979,pp. 75-81.

Hill(1979) D. Hill, W.M. vanCleemput, "SABLE: A tool for generating structured, multi-level simulations", Proc. 16th Design Automation Conference (ACM/IEEE), San Diego, California, 1979, pp. 272-279.

Hwang(1982) K.Hwang and Y.-H. Cheng, "Partitioned algorithms for VLSI Arithmetic Systems", IEEE Trans. on Computers, Dec. 1982, Vol. C-31, No.12, pp 1215-1224.

INMOS(82) "HDL: A hardware description language", in "Advanced Course in VLSI Architecture", University of Bristol (UK), July 1982.

INMOS(82a) "OCCAM, Abstract, Multiprocessor applications, Semantics, Design station presentation", INMOS, see also Electronics Nov./Dec. 1982.

Kramer/Leeuwen(82) M.R. Kramer, J. van Leeuwen, "Systolic Computation and VLSI", RUU-CS-82-9, June 82, University of Utrecht.

Kung(79) H.T. Kung, "Let's design algorithms for VLSI systems", Dep. of Computer Science, Carnegie Mellon University, Pittsburgh, Penn.

Kung(1980) H.T. Kung, "The structure of Parallel Algorithms", Advances in Computers, Vol. 19, 1980.

Lattice(82) "MODEL: A high-level structured design language", in "Designing with Gate Arrays", Lattice Logic Ltd, Edinburgh, 1982.

Leiserson/Saxe (81) C.E. Leiserson, J.B. Saxe, "Optimizing synchronous systems", Proc. 22nd Annual Symposium on Foundations of Computer Science, IEEE, October 1981.

Lengauer(82) Thomas Lengauer, "Eine Spezifikationssprache für integrierte Schaltkreise", A 82/7 Universität des Saarlandes.

Levitt(72) K.N. Levitt, W.H. Kautz, "Cellular Arrays for the solution of graph

problems, Communications of the ACM, Sept.72, Vol. 15, No.9, pp 789-801.

Lim(82) W. Y-P. Lim, "HISDL - A structure description language", Comm. ACM, Vol. 25, No. 11, pp 823-830.

Lipton(82) R.J.Lipton, R.Sedgewick, J.Valdes : Programming Aspects of VLSI, 9th ACM POPL Symposium, Jan 1982, pp 57-65.

Mead/Conway(80) C. Mead, L. Conway, "Introduction to VLSI Systems", Addison-Wesley, 1980.

Mehlhorn(82) K. Mehlhorn, St. Näher, M. Novak, HILLSIM, Ein Simulator für MOS-Schaltkreise, Universität des Saarlandes, August 1982.

Misunas(1973) D. Misunas, "Petri Nets and Speed Independent Design", CACM, Vol 16, No. 8, pp 474-481, Aug. 1973.

Moldovan(1982) D.I. Moldovan, "On the analysis and synthesis of VLSI algorithms", IEEE Transactions on Computers, Nov. 1982, Vol. C-31, No 11, pp 1121-1125.

Nestor/Thomas(1982) J.A. Nestor, D.E. Thomas, "Defining and Implementing a Multilevel Design Representation With Simulation Applications", 19th Design Automation Conference, pp. 740-746, Las Vegas 1982.

Newkirk(82) J. Newkirk, Rob. Mathews, Peter Eichenberger, Dan Perkins, "A constraint-based layout description system", Stanford University, VLSI report.

Nievergelt et al. (82) G. Beretta, H. Burkhart, P. Fink, J. Nievergelt, J. Stelovsky, H. Sugaya, A. Ventura, J. Weydert, " XS-1: An integrated interactive system and its kernel", 6th Intern. Conf. on Software Engineering, Tokyo, Japan, 1982.

Organick (79) G.F. Maxey, E.I Organick, "CASL - A Language for Automating the Implementation of Computer Architectures", 4th International Symposium on Computer Hardware Description Languages and Their Applications, Palo Alto, California (ACM/IEEE), 1979, pp. 102-108.

Ottmann(82) T.A. Ottmann, A.L. Rosenberg, L.J. Stockmeyer, "A dictionary machine for VLSI", IEEE Transactions on Computers, Sept. 82, Vol. C-31, No.9, pp. 892-897.

Reed(1952) I.S. Reed, "Symbolic synthesis of digital computers", Proc. ACM, Toronto 1952, pp. 90-94.

Reed(1953) I.S. Reed, "Symbolic design of digital computers", MIT Lincoln Laboratory Technical Memorandum no. 23, Lexington, Mass. 1953.

Rem(1981) M. Rem, C. Mead (1981), "A notation for designing restoring logic circuitry in CMOS", Proc. 2nd Caltech Conference on VLSI, ed. C.L. Seitz, California Institute of Technology, Pasadena, Calif.

Rem(1982) M.Rem, "On the design of restoring logic circuitry", Advanced

course on VLSI architecture, University of Bristol.

Rivest(1982) R. Rivest, "The PI System", Proc. Design Automation Conference, Las Vegas, 1982.

Robinson(82) P. Robinson, J. Dion, "Design Aids for Uncommitted Logic Arrays", Cambridge University Computer Laboratory, Cambridge, England.

Rosenfeld(83) A. Rosenfeld, "Parallel Image Processing Using Cellular Arrays", Computer, Vol. 16, No.1, Jan. 83, pp 14-22.

Sakai(1982) T. Sakai et al., "An interactive simulation system for structured logic design -- ISS", 19th Design Automation Conference, pp 747-754, Las Vegas 1982.

Sastri (82) S. Sastri, S.Klein, "PLATES: A metric-free layout language", Proc. Conference on Advanced research in VLSI, P.Penfield Editor, MIT, Cambridge, January 1982.

Sequin(82) C.H. Sequin, "Managing VLSI Complexity", Draft of invited paper for IEEE proceedings.

Sequin(1981) C.H. Sequin, "Standard interchange formats for integrated circuit design", Computer Science Division, UC Berkeley.

Siskind(1982) J.M. Siskind, J.R. Southard, and K.W. Crouch, "Generating Custom High Performance VLSI Designs from Succinct Algorithmic Descriptions", Proc. Conference on Advanced Research in VLSI, P. Penfield, Editor, January (1982), pp 28-40.

Suzuki(1982) N. Suzuki, R. Burstall, "Sakura: A VLSI modeling language", Proc. Conf. on Advanced Research in VLSI, MIT, 1982.

Thompson (1981) C.D. Thompson, "VLSI Complexity of Sorting", CMU Conference on VLSI Systems and Computations, pp. 108-118.

Trimberger(1981a) S. Trimberger, J.A. Rowson, C.R. Lang, J.P. Gray, "A structured design methodology and associated software tools", IEEE Transactions on Circuits and Systems, Vol. CAS-28, No.7, July 1981.

Trimberger(1981b) S. Trimberger, "Combining graphics and a layout language in a single interactive system", 18th Design Automation Conference, 1981.

Valdes(1982) J. Valdes, ALI2 Documentation and Implementation Guide: Language Overview, Version 2, Dep. of EECS, Princeton University.

vanCleemput(79) W.M. vanCleemput, "Computer Hardware Description Languages and their Applications", Proceedings 16th Design Automation Conference, June 1979.

Weste(81) N.Weste, B. Ackland, "A pragmatic approach to topological symbolic IC design", VLSI 81 Proc., pp. 117-130.

Wirth(1981) N. Wirth, "The personal computer Lilith", Rep. 40, Inst. für

Informatik, ETH Zürich, April 1981.

Wirth(1982) N. Wirth, "Hades: A Notation for the Description of Hardware", ETH Zurich, August 1982.

Wirth(1982a) N. Wirth, "Programming in Modula-2", Springer Verlag 1982.

Berichte des Instituts für Informatik

*Nr.25	U. Ammann:	Error Recovery in Recursive Descent Parsers and Run-time Storage Organization
Nr.26	E. Zachos:	Kombinatorische Logik und S-Terme
*Nr.27	N. Wirth:	MODULA-2
*Nr.28	J. Nievergelt, J. Weydert:	Sites, Modes and Trails: Telling the User of an Interactive System where he is, what he can do, and how to get to places
*Nr.29	A.C. Shaw:	On the Specification of Graphic Command Languages and their Processors
*Nr.30	B. Thurnherr, C.A. Zehnder:	Global Data Base Aspects, Consequences for the Relational Model and a Conceptual Schema Language
*Nr.31	A.C. Shaw:	Software Specification Languages based on regular Expressions
Nr.32	E. Engeler:	Algebras and Combinators
*Nr.33	N. Wirth:	A Collection of PASCAL Programs
*Nr.34	R. Marti, J. Rebsamen, B. Thurnherr:	Meta Data Base Design - Consistent Description of a Data Base Management System
*Nr.35	H.H. Nägeli, R.Schoenberger:	Preventing Storage Overflows in High-level Languages
	J. Hoppe:	A Simple Nucleus written in Modula-2
*Nr.36	N. Wirth:	MODULA-2 (second edition)
Nr.37	Hp. Bürkler, C.A. Zehnder:	EDV-Projektentwicklung - Ein Arbeitsheft für Informatik-Studenten
*Nr.38	H. Burkhart, J. Nievergelt:	Structure-oriented editors
*Nr.39	A. Meier, C.A. Zehnder:	Flächenmodell-Register: Die Strukturen wichtiger geographischer Datensammlungen der Schweiz
*Nr.40	N. Wirth:	The Personal Computer Lilith
Nr.41	T.M. Fehlmann:	Theorie und Anwendung des Graphmodells der Kombinatorischen Logik
Nr.42	E. Graf:	Probabilistische Algorithmen und Computer-unterstützte Untersuchungen von probabilistischen Primalitätstests
*Nr.43	H. Burkhart:	Konzepte zur Systematisierung der Benutzerschnittstelle in interaktiven

		Systemen und ihre Anwendung auf den Entwurf von Editoren
*Nr.44	J. Nievergelt, F.P. Preparata:	Plane-sweep Algorithms for Intersecting Geometric Figures
Nr.45	M. Reimer, J.W. Schmidt:	Transaction Procedures with Relational Parameters
Nr.46	J. Nievergelt, H.Hinterberger, K.C. Sevcik:	The Grid File: An adaptable, symmetric multi-key file structure
Nr.47	J. Nievergelt:	Errors in dialog design and how to avoid them
Nr.48	P. Läuchli:	PG - Ein interaktives System für die Manipulation von Figuren der projektiven Geometrie
Nr.49	A. Meier:	A Graph Grammar Approach to Geographic Data Bases
Nr.50	J. Rebsamen, M. Reimer,	LIDAS A Database System for the Personal Computer Lilith
	P. Ursprung, C.A. Zehnder:	The Database Management
Nr. 51	K.J.Lieberherr, S.E. Knudsen:	Zeus: A Hardware Description Language for VLSI

* out of stock