# USE OF A THEOREM PROVER FOR TRANSFORMATIONAL SYNTHESIS

A.M. KOELMANS
Computing Laboratory
University of Newcastle upon Tyne
NE1 7RU United Kingdom

F.P. BURNS
D.J. KINNIMENT
Department of Electrical and Electronic Engineering
University of Newcastle upon Tyne
NE1 7RU United Kingdom

### Abstract

Transformational synthesis is the process of generating a hardware implementation from an initial behavioural description, by repeatedly applying transformations to the behavioural descriptions until a satisfactory implementation can be generated. It is essential to verify the correctness of the applied transformations if the final implementation is to conform to the initial specification. We have implemented a prototype interactive design tool that integrates the Boyer Moore theorem prover into the design process. We describe the reasons for using this particular theorem prover, its basic features, and demonstrate the operation of the design tool and its interaction with the theorem prover with an example.

## 1 Introduction

The use of formal methods in VLSI design is currently a very active research area. This requires the specification of the behaviour of a hardware design in a formal, mathematically rigorous manner. Such specifications can subsequently be manipulated by proof systems (usually called 'proof assistants' or 'theorem provers') to prove the equivalence of hierarchical or temporal properties of hardware designs.

The development of hardware synthesis systems seems to have taken place without the involvement of formal methods. The problem in hardware synthesis is to take an initial specification, frequently in the form of boolean equations or register transfer statements, and map these efficiently into silicon. Such languages do not lend themselves very well to formal reasoning, and recent research efforts have only concentrated on mapping formal languages into silicon. The use of formal languages makes it difficult to generate an efficient layout, since a much larger design space needs to be explored than when low level languages are used. This means that formal specifications will usually need to be transformed, in several stages, to a low level form which can then be mapped into silicon. A major problem here is to ensure that the applied transformations do not invalidate the original specification. This, of course, should involve the use of a proof system, and this paper describes our efforts in this area.

We describe a prototype tool which integrates a theorem prover into the design environment in such a way as to ensure functional and transformational correctness at all times. The system is driven from a hardware description language developed at Newcastle called STRICT, and it is interactive. Our aim in developing this tool is to make it easier for the designer to ensure correctness of the final implementation, without losing the benefit of his skill and experience.

There are two proof systems which have already shown great promise in the area of hardware design: HOL [Go87] and the Boyer Moore theorem prover [Bo88]. Both use a specification language based on formal semantics, have been used to verify substantial hardware designs, are widely used in the VLSI research community, and are readily available. HOL can be most accurately described as a proof assistant, that is, it requires interactive commands from a user who is an expert in the internal workings of the system. The Boyer Moore theorem prover requires all its input from a text file, and runs without interactive user input during the proof process. For this reason, we felt that this prover is a more suitable tool in a hardware synthesis environment, and we therefore decided to use it in conjuction with our synthesis tool.

The paper is organised as follows. In section 2 we introduce the basic features of high level synthesis systems. In section 3 we describe the Boyer Moore theorem prover, and in section 4 an example of its use is presented. In section 5 we describe the basic features of our tool, and we finally present an example of its operation in section 6.

## 2 Transformational synthesis

An excellent introduction to high level synthesis systems can be found in the overview paper by McFarland [Mc88]. All high level synthesis systems initially operate upon a user specified behavioural description in an appropriate high level language. The language we use for this purpose is called STRICT [Ca85], which is a conventional hardware description language which allows specifications to be written in a functional notation. This description is parsed into an internal format. This is followed by the scheduling and allocation phase, during which the basic functional units of the design are determined and the basic hardware units are assigned to these functional units, together with memory elements and communication paths. The resulting design is then fed into conventional floorplanning and routing tools to produce the final chip layout. State of the art examples of high level synthesis systems are Cathedral [De86] and the Yorktown Silicon Compiler [Br88].

The high level synthesis community is split into two camps: those who want to fully automate the design process, and those who want to make use of the designer's experience by providing interactive input during the design process. In fully automatic systems, much of the current reseach is focussed on the provision of algorithms to ensure that reasonably efficient hardware is generated from a large number of possible implementations.The interactive approach encourages designers to investigate different designs through experimenting with different architectures generated from the same specification by, for instance, serialising operations upon the same device or by allocating many operations to devices operating in parallel. We firmly believe in the interactive approach, and the tool described in this paper was inspired by SAGE [De88], a sophisticated interactive synthesis system, but one that does not incorporate verification features.

A specification of an algorithm in a high level language will in general be unsuitable for direct translation into silicon. It will be necessary to perform transformations which preserve the behaviour, but generate a much more compact silicon implementation [Mc88]. The problem is then to ensure that the applied transformations preserve the behaviour [Ca88], and in general to verify the correctness of the ultimate design produced by the system against the initial specification. Research in this area is reported in [Ve88, El89,Fi89]. As mentioned in the previous section, we use the Boyer Moore theorem prover to achieve this goal.

## 3 The Boyer–Moore Theorem Prover

The Boyer Moore theorem prover [Bo88] was initially developed at the University of Austin as a tool for research in Artificial Intelligence applications. It has been successfully applied in such diverse areas as list processing, number theory, protocols, real time control, and concurrency. The largest proof to date performed by the prover is that of Goedel's Incompleteness Theorem (it is distributed with the source code of the prover). It was used by Hunt [Hu85] to verify the correctness of a microprocessor design, thereby confirming that this prover could be used in the VLSI design area. The prover is written in Common Lisp, and is distributed with a specially modified version of the Kyoto Common Lisp interpreter. The software runs on a wide range of machines.

The Boyer Moore logic is a quantifier free, first order logic with equality and function symbols. The input language to the prover is a variant of Pure Lisp. This input language is used to present the prover with a sequence of so called events. The most important events are: definitions of (often recursive) functions to be used in proofs, and theorems. The keyword DEFN starts the description of a function; all functions must conform to certain rules with respect to termination, which are checked by the prover before accepting the function definition. If a function calls other functions, these must have been previously defined. The keyword PROVE–LEMMA initiates a description of a theorem. A theorem will be acceptable if it can be proven by the prover. The prover will attempt to do this using its built–in logic inference rules, definitions, and previously proved theorems. It will in general be necessary to build a database of carefully chosen and carefully ordered theorems to perform complex proofs.

As an example, we can define a function DOUBLE which doubles an integer by adding the integer to itself, and we then prove that this is the same as multiplying by 2:

```
(defn double (x)
   (plus x x))

(prove-lemma double-is-times-2 (rewrite)
  (implies (numberp x)
    (equal (double x) (times 2 x))
  )
)
```

The theorem includes the hypothesis that x must be a number, and requires it to be stored as a rewrite rule. After it has been proved, the prover will subsequently replace occurences of (double x) by (times 2 x).

The prover has a number of powerful built–in heuristics, one of which is the capability of performing mathematical induction. This means it can cope most efficiently with recursively defined functions (which are not necessarily the computationally most efficient). The prover is also distributed with a number of databases of useful functions and theorems which it frequently will use in proofs. These can be loaded at the request of the user.

The prover has a facility whereby the user can define his own abstract data types. This is usually referred to as the *Shell principle*. For the purpose of hardware verification, Hunt [Hu85] has added a data type for use with bit vectors, and a whole set of associated theorems. These can be loaded automatically when the prover is started.

If the prover fails to complete a proof, this will not necessarily mean that there is something wrong with the theorem. It may be that if the proof is rather complex, the prover will not 'see' the right path through the proof. In this case it will be necessary to 'educate' the prover by first proving smaller theorems which prove parts of the big proof, so that the prover will subsequently use these to complete the big proof. In many cases these smaller theorems will be useful in themselves, and can be added to the users' database if he wishes. The prover will therefore frequently need help from the user, which means that in a sense it is not fully automatic.

The prove–lemma construct can have as an optional parameter a list of hints, which can tell the prover to use certain theorems with certain parameters, ignore other theorems, or perform inductions in a specified manner. Some of these features are shown in the example which now follows.

# 4 A Proof Example

We want to prove that for all positive n the integer quotient of n*x and n*y is equal to the integer quotient of x and y, ie. we will try to prove that

```
(equal (quotient (times n x)
                 (times n y))
       (quotient x
                y))
```

Such a theorem occurs frequently when trying to prove properties of bit vectors, with n having the value 2. We will call this theorem QUOTIENT–TIMES–TIMES. Trying to prove it without the aid of additional theorems fails – the prover tries to solve the problem by using induction, which is not the right approach.

All proof times mentioned below are as measured by the prover on a Sparcstation. A substantial portion of Hunt's database was loaded first. The recursive definition of QUOTIENT is [Bo88]:

```
(defn quotient (i j)
   (if (equal j 0) 0
        (if (lessp i j) 0
            (add1 (quotient (difference (i j) j)))
```

That is, the quotient is performed by repeated subtraction, while counting the number of subtractions performed.

The proof can be split up in three cases: x < y, x == y and x > y. The first case is described by the following lemma:

```
(prove-lemma q1 (rewrite)
  (implies (and (greaterp y x)
                (greaterp n 0)
           )
    (equal (quotient (times n x) (times n y))
           (quotient x y)
    )
  )
)
```

The prover's builtin arithmetic rules are not sophisticated enough to cope with this, so we first prove another lemma to tell it that if x < y then n*x < n*y:

```
(prove-lemma q1-first (rewrite)
  (implies (and (greaterp y x)
                (greaterp n 0)
           )
    (greaterp (times n y) (times n x)
    )
  )
)
```

The proof takes 6 seconds. Now lemma q1 is proved in 9 seconds, using q1–first and the prover's builtin rules.The case x == y is easily proved from builtin rules. It takes just two seconds:

```
(prove-lemma q2 (rewrite)
  (implies (and (equal y x)
                (greaterp n 0)
           )
     (equal (quotient (times n x) (times n y)
            (quotient x y)
```

```
      )
   )
)
```

The third case, x > y, is the most general one and fails if tried on its own. It can be proven if one realises that if x > y, then x = a*y + b, where a >0, and b < y. Thus we first try the following lemma:

```
(prove-lemma q3-first (rewrite)
  (implies (and (greaterp a 0)
                (greaterp y b)
                (greaterp n 0)
                (equal (plus (times a y) b) x)
           )
    (equal (quotient (times n x) (times n y))
           (quotient x y)
    )
  )
)
```

The proof of this lemma is entirely achieved by simplification using various database rules, and takes 7 seconds.

In order to prove the lemma for the case x > y we need to tell the prover to use this lemma, while substituting (quotient x y) for a, and (remainder x y) for b. This will allow the prover to check that the hypothesis (greaterp x y) holds. Note the use of a hint in this lemma:

```
(prove-lemma q3 (rewrite)
  (implies (and (greaterp x y)
                (greaterp n 0)
           )
    (equal (quotient (times n x) (times n y))
           (quotient x y)
    )
  )
  ((use (q3-first (a (quotient x y))
                  (b (remainder x y))
        )
   )
  )
)
```

The proof takes 40 seconds. The main lemma still cannot be proven – we have to tell the prover that the previous three cases together constitute any possible combination of x and y:

```
(prove-lemma q-bridge (rewrite)
  (implies (and (greaterp n 0)
                (or (greaterp y x)
                    (equal y x)
                    (greaterp x y)
                )
           )
     (equal (quotient (times n x) (times n y))
            (quotient x y)
     )
  )
  ((use (q1))
   (use (q2))
   (use (q3))
```

```
  )
)
```

This proof takes 6 seconds. We are now ready to do the final proof, which takes 2 seconds.

```
(prove-lemma quotient-times-times (rewrite)
  (implies (and (greaterp n 0)
                (numberp x)
                (numberp y)
           )
     (equal (quotient (times n x) (times n y))
            (quotient x y)
     )
  )
  ((use (q-bridge)))
)
```

Afer this proof has been completed, all but the last lemma should be disabled, since they are only a special case of quotient–times–times. In other proof efforts, some of the lemmas may be quite useful, and should therefore be kept. [Bo88] contains several examples.

# 5 Synthesis method

The specification to be used in the synthesis process must already have been written in the STRICT language [Ca85], a proprietary hardware description language similar to VHDL [IE87]. In STRICT, the behavioural specification for each block is captured in a set of functional expressions, along with temporal information. The tool transforms this specification into a functional tree. This tree is drawn on a screen by a graphical subsystem with which the designer can subsequently interact. In order to ensure that all interactions preserve the correctness of the design, only changes that correspond to a set of formal transformations are allowed, and these must first have been verified by the Boyer Moore theorem prover .

These transformations are kept in two libraries, one generated by the user of the tool (for use with the current user defined specification), and the other one a standard library of rewrite rules which can be applied to the set of operators built in to the STRICT language. Both libraries come in the form of a text file, and their contents must be acceptable to the Boyer Moore prover (it is the user's responsibility to ensure that they are). Both libraries are therefore generated offline, before they synthesis tool can be used. The standard library contains rules relating to the following STRICT operators:

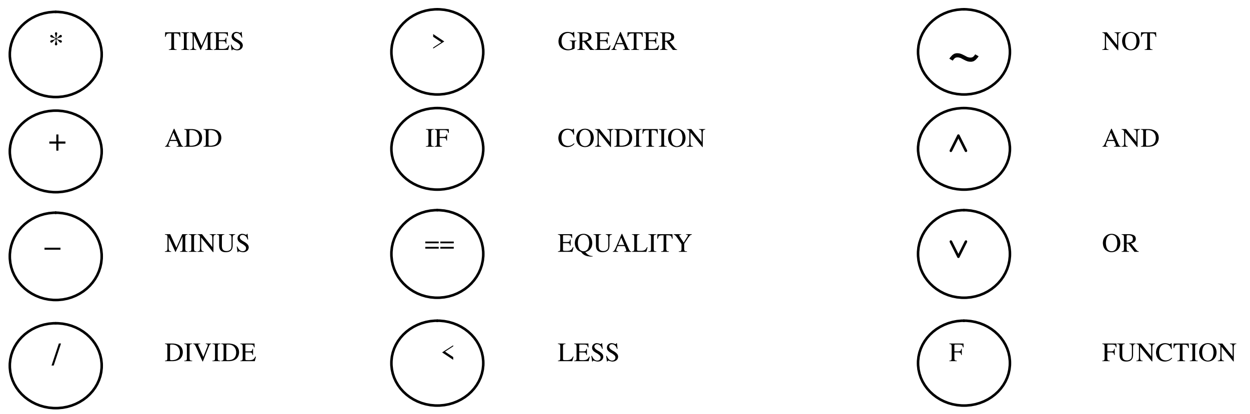



Fig 1

The library is arranged in sections where each section corresponds to one of the nodes shown in Fig. 1. If, for example, the designer clicks on the + node on the screen, the rewrite rules relating to the add node are made available. A typical example of such a rule might be

```
(equal (plus a (plus b c))
       (plus a b c))
```

which simplifies the functional tree by removing one plus operation.

A behavioural specification in a high level language will usually not correspond to the most efficient hardware implementation. It will generally be necessary to modify its functional tree in order to improve the efficiency. Since only changes can be made that have first been verified by the prover , all modifications are by definition correct. The theorem prover therefore ensures that all changes made are carried out within a formal framework. Once the designer has completed his work on the functional tree, hardware can be allocated, by fetching the appropriate modules corresponding to the various parts of the design from a hardware library.

When the transformations and hardware allocation have been completed, a STRICT description of the complete design is generated, and the final layout can then be generated using standard floorplanning and routing tools.

The interactive interface of the synthesis tool is shown in fig. 2. The functional tree for one level in the design hierarchy is displayed in the centre of the screen. Below it is its lisp description. Interaction with the functional tree takes place by clicking on the icons which are situated around the edges of the screen. To access the subtrees of the functional tree, a small list of icons on the right hand side of the screen is available. It begins with eva and ends in ret (which stands for return, and which enables the user to move back up the design hierarchy). To carry out modifications to the tree the THSRCH icon at the bottom left of the screen is selected, followed by a node on the functional tree. The system responds by searching the rewrite rule library section associated with the chosen node and selects a list of applicable rewrite rules which are presented to the user as options. If the user wishes to carry out the modification, he selects it directly and applies the change. The STORE icon which is above the THSRCH icon can be used to store a particular rule which the designer may wish to apply more than once. In this case the rule is passed to a small buffer where it can be selected and applied without searching the library for it. The REWRITE icon at the bottom right hand corner of the screen is used for rewriting a subtree within a functional tree. The ALLOC and DEALLOC icons are used for the purpose of allocating hardware to the functional tree. Allocation for a particular node on the functional tree is done by clicking on the ALLOC icon followed by the selection of a node on the functional tree. The system responds by searching a library of hardware modules each of which has an associated behavioural description. A list of modules with area and time information is provided which are guaranteed to implement the behaviour at the chosen node. The user then selects a module, and the functional tree is modified at the node where the implementation is carried out by exchanging it for a new node representing the hardware module. To deallocate, the DEALLOC icon is selected followed by a previously allocated node. In this case the node is replaced by the subtree representing the behaviour of the particular hardware module. Allocation of a functional tree can be carried out automatically by clicking on the AUTOALL icon. This causes the tool to search the library of hardware modules and map them directly to the functional tree. The MRG/SPL icon is used for space–time transformations. Finally the EXIT icon is used for exiting from the synthesis tool.

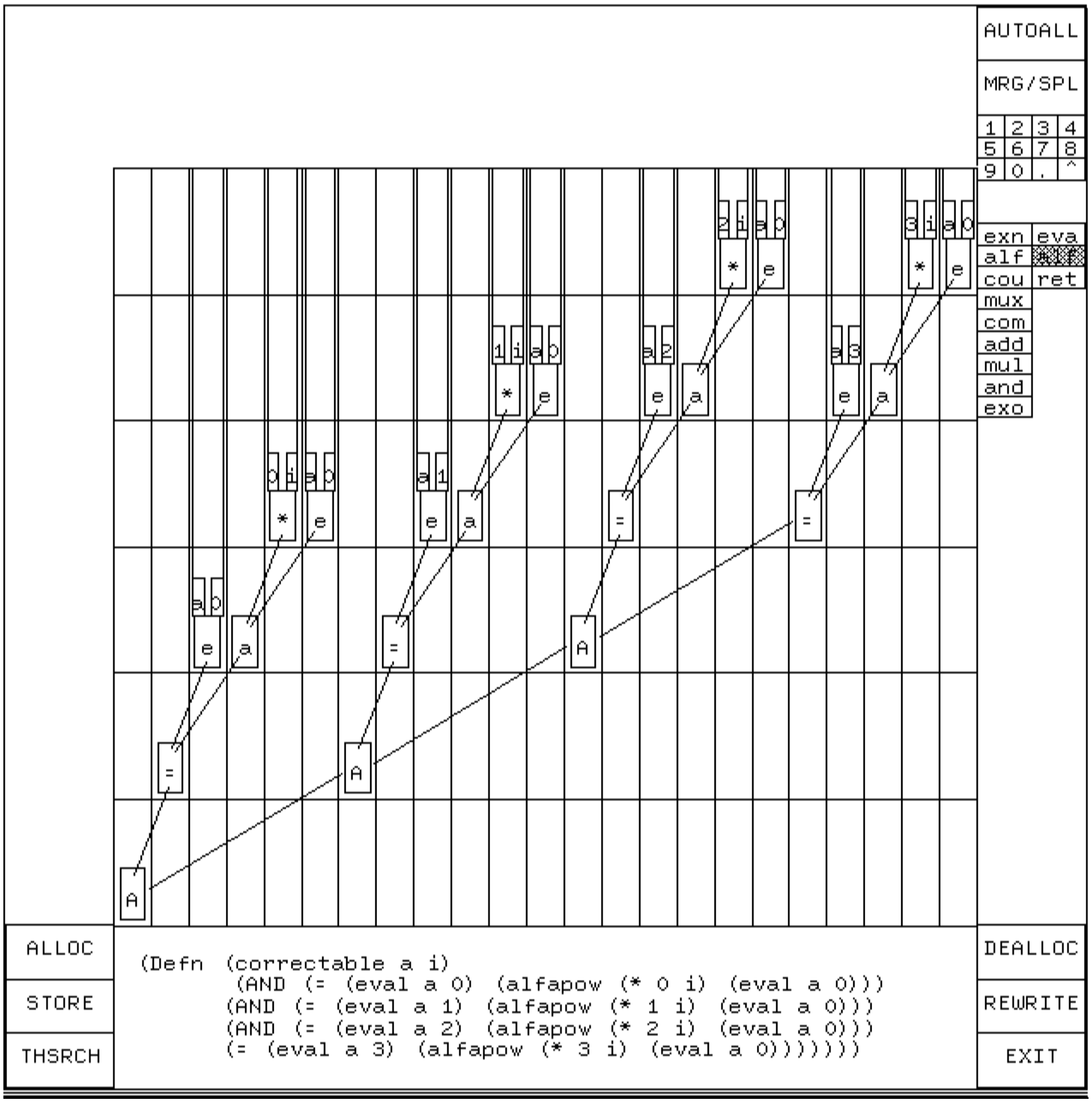



Fig. 2

To apply a change to a particular point, a node from the tree is selected. The change must be in the form of a Boyer Moore rewrite rule, selected from one of the available libraries. The left hand side of this rule is matched against the tree from the chosen node (in the upward direction). If a match is found the change will then be applied. The right hand side of the rewrite rule containing the new structure is substituted in place of the old structure, and all connecting nodes are appended to the new section of tree.

Some designs may be of a regular nature and have a repetitive structure which can only be modified efficiently by applying a change repetitively throughout the design. For this purpose a feature is provided which allows changes to be made globally throughout the design.

Changes made to the original functional tree by the designer are recorded, by storing them in a text file. This is useful for a number of reasons. It enables the designer to check upon his own modifications once the design is completed. Also, whilst changes are being made to the original specification by applying rewrite rules, the theorem prover may generate new rules which the designer may wish to keep. For example, a larger rewrite rule may result from a series of smaller changes, or the designer may be able to derive a new rewrite rule. If such a rewrite rule is stored, it could easily be added to the user library at a later point.

Allocation of hardware is split into two stages, manual and automatic. The manual stage concerns the binding of operational units to the operators on the functional tree. For this purpose a library of hardware components is available, so that the designer can choose from a possibly large set Area and time information is shown in graphical form at the top of the screen as he chooses his components and allocates them.

# 6 Synthesis example

The screen of Fig. 2 shows an example function which was taken from a error decoder module described by Kalker [Ka88]. The function is called 'correctable' and would be defined in STRICT as follows:

```
correctable(a: byte[32], i: integer):BOOLEAN ::=
     (eval(a, 0) == alfapow(0*i, eval(a, 0))) AND
     (eval(a, 1) == alfapow(1*i, eval(a, 0))) AND
     (eval(a, 2) == alfapow(2*i, eval(a, 0))) AND
     (eval(a, 3) == alfapow(3*i, eval(a, 0)))
```

The bottom of the screen shows the S–expression equivalent of this function. We demonstrate the operation of the tool by showing how one can make this function more efficient by making formal transformations.

Any modifications that might be applied to the tree rely upon the use of rewrite rules from the library, such as:

```
1 /  (equal ((times i 0) 0))
2 /  (equal (equal a a) t))
3 /  (equal ((and t a) a))
4 /  (equal (and a (and b c))
            (and (and a b) c)))
```

In the rest of this section we will frequently refer back to these rewrite rules.

We turn our attention to the leftmost subtree on the screen of fig. 2. After clicking on the multiplication node, the screen of fig. 3 appears.

Near the bottom of the screen 3 boxes have appeared, indicating the fact that three rewrite rules from the library are applicable at this point. The first of these rules is printed below the boxes, and by clicking on each of the boxes in turn, the designer can cycle through them. Application of rule 1 (which says that 0*i equals 0) will result in deletion of the multiplication subtree. This is shown in fig. 4. We then have a subtree representing alfapow(0,eval(a,0)), which can be replaced by eval(a,0) by rewriting. The rewrite rule is shown at the bottom of fig. 5, and the result of the application of the rule in fig. 6. The resulting subtree has two equal branches, so rule 2 can be applied (see fig. 7) to the = node to give the result T. This is shown in fig. 8. Since ANDing with T is a no–op operation (see bottom of fig. 9), rule 3 can be applied to the AND node, resulting in the removal of the leftmost subtree.

The final functional tree after the above changes have been applied is shown in fig. 10. The leftmost subtree has disappeared completely as a result of the transformations applied, thereby producing a more efficient design. The updated version of the 'correctable' function is displayed near the bottom of the screen.

At this point, it will be possible for the designer to allocate actual hardware. This is shown in fig. 11, where nodes that have been allocated are shown as shaded boxes. At the top of the screen, for each box an estimate of the area and the speed for each node is shown. These in turn may prompt the designer to select a particular node, for example, one with a very large area, in order to do further optimisation.

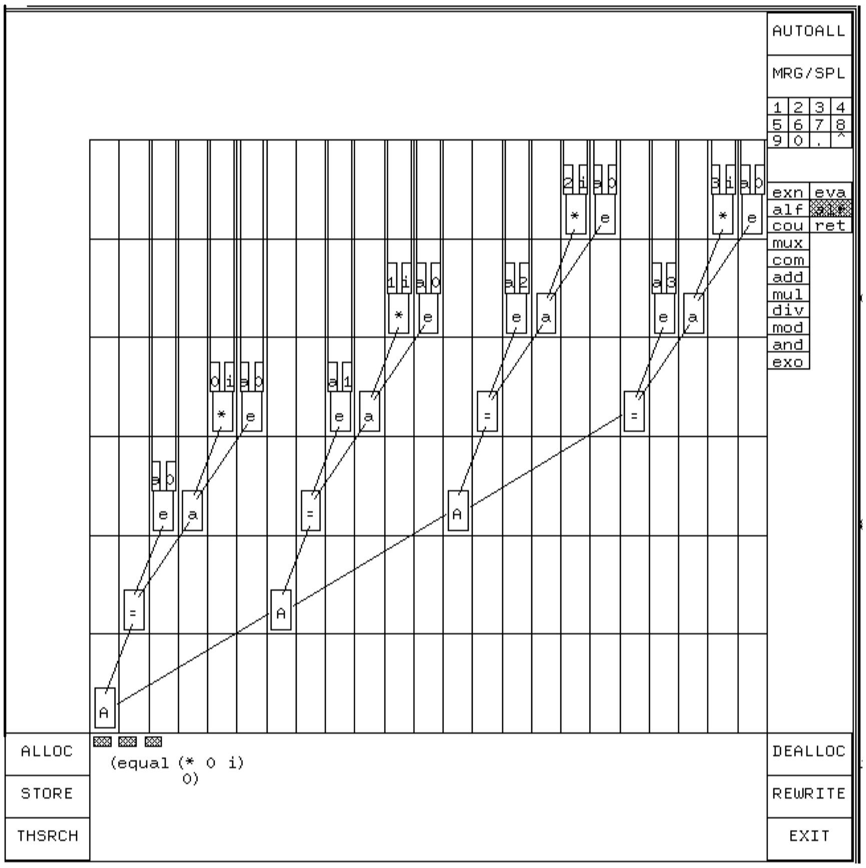

AUTOALL
MRG/SPL
1 2 3 4
5 6 7 8
9 0 . ^
exn eva
alf
cou ret
mux
com
add
mul
div
mod
and
exo
ALLOC
STORE
THSRCH
(equal (* 0 i)
0)
DEALLOC
REWRITE
EXIT

Fig. 3

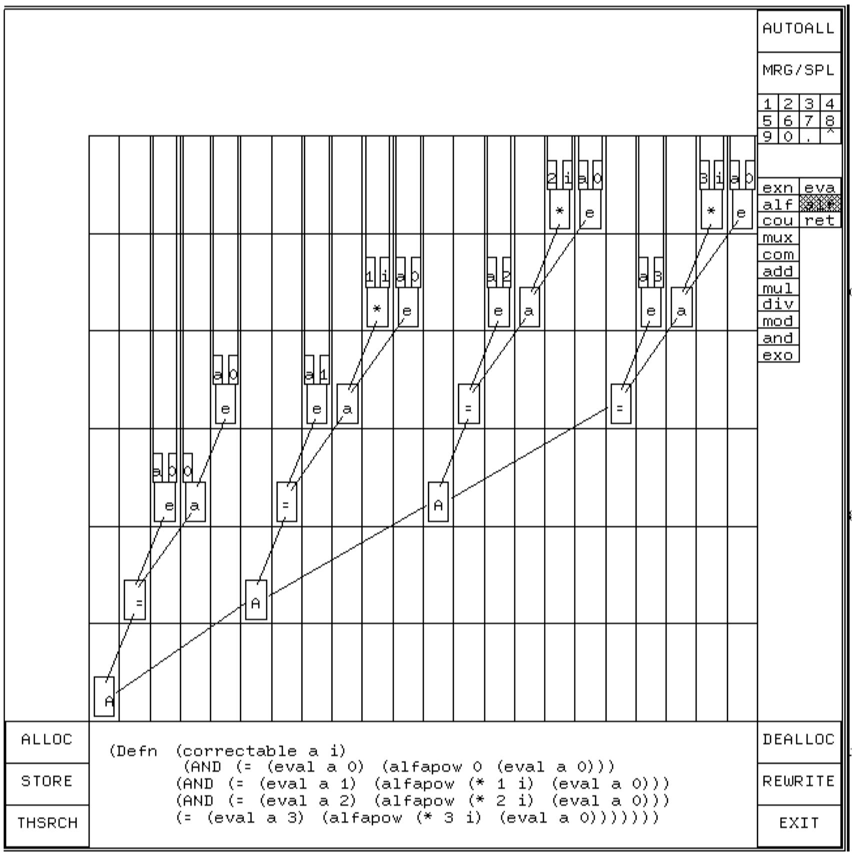

AUTOALL
MRG/SPL
1 2 3 4
5 6 7 8
9 0 . ^
exn eva
alf
cou ret
mux
com
add
mul
div
mod
and
exo
ALLOC
STORE
THSRCH
(Defn (correctable a i)
(AND (= (eval a 0) (alfapow 0 (eval a 0)))
(AND (= (eval a 1) (alfapow (* 1 i) (eval a 0)))
(AND (= (eval a 2) (alfapow (* 2 i) (eval a 0)))
(= (eval a 3) (alfapow (* 3 i) (eval a 0)))))))
DEALLOC
REWRITE
EXIT

Fig. 4

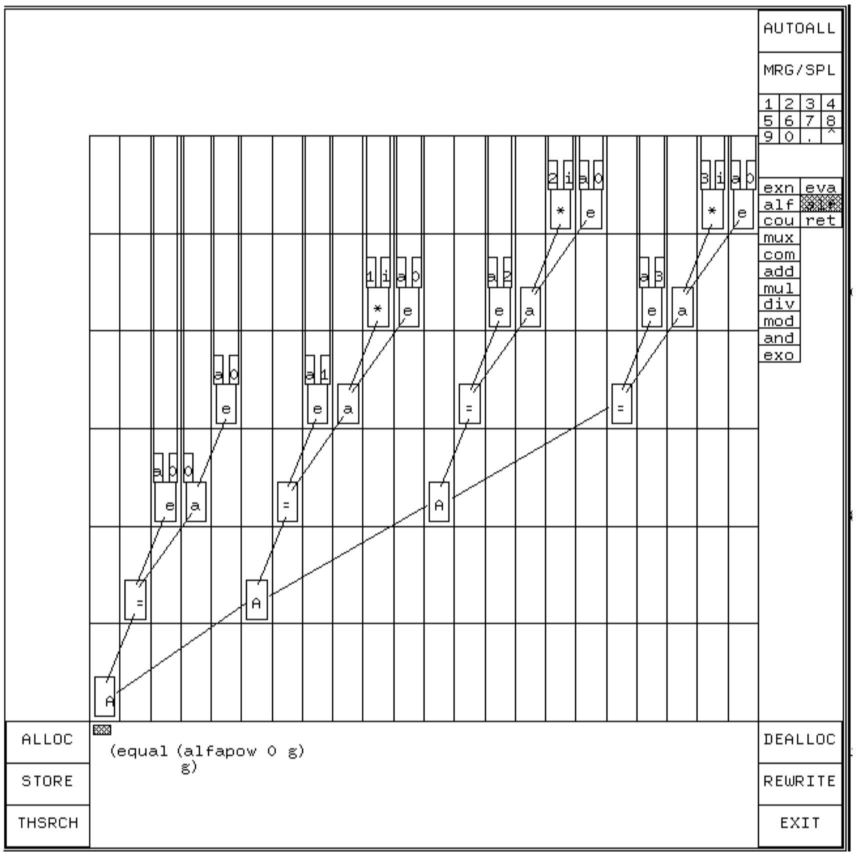

AUTOALL
MRG/SPL
1 2 3 4
5 6 7 8
9 0 . ^
exn eva
alf
cou ret
mux
com
add
mul
div
mod
and
exo
ALLOC
STORE
THSRCH
DEALLOC
REWRITE
EXIT
(equal (alfapow 0 g)
g)

Fig. 5

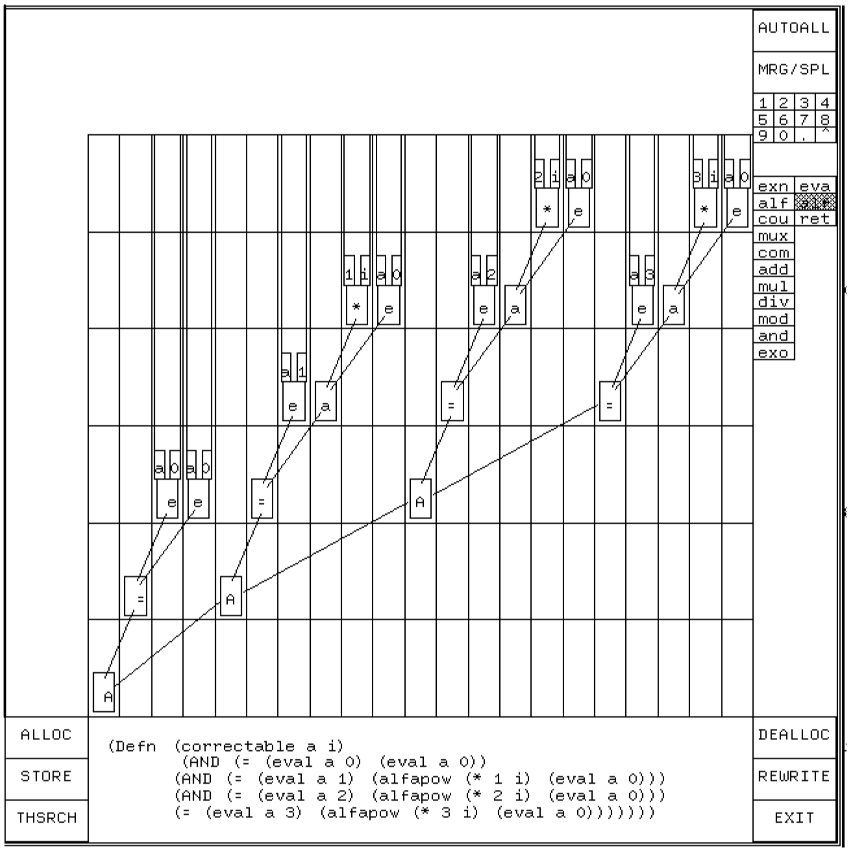

AUTOALL
MRG/SPL
1 2 3 4
5 6 7 8
9 0 . ^
exn eva
alf
cou ret
mux
com
add
mul
div
mod
and
exo
ALLOC
STORE
THSRCH
DEALLOC
REWRITE
EXIT
(Defn (correctable a i)
(AND (= (eval a 0) (eval a 0))
(AND (= (eval a 1) (alfapow (* 1 i) (eval a 0)))
(AND (= (eval a 2) (alfapow (* 2 i) (eval a 0)))
(= (eval a 3) (alfapow (* 3 i) (eval a 0)))))))

Fig. 6

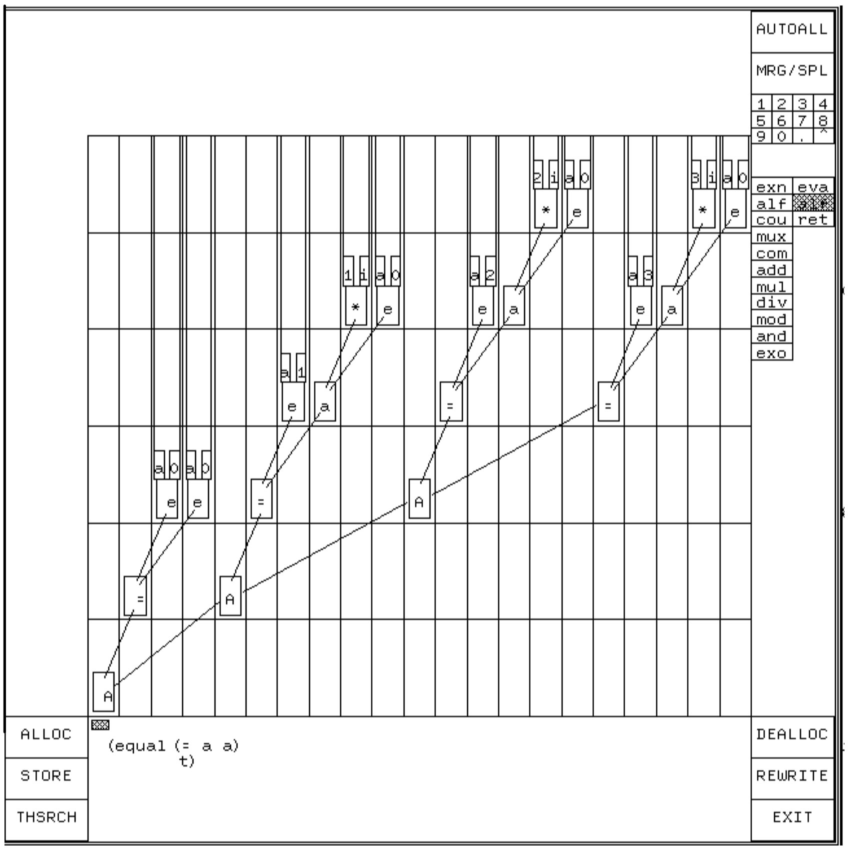

AUTOALL
MRG/SPL
1 2 3 4
5 6 7 8
9 0 . ^
exn eva
alf
cou ret
mux
com
add
mul
div
mod
and
exo
ALLOC
STORE
THSRCH
(equal (= a a)
t)
DEALLOC
REWRITE
EXIT

Fig. 7

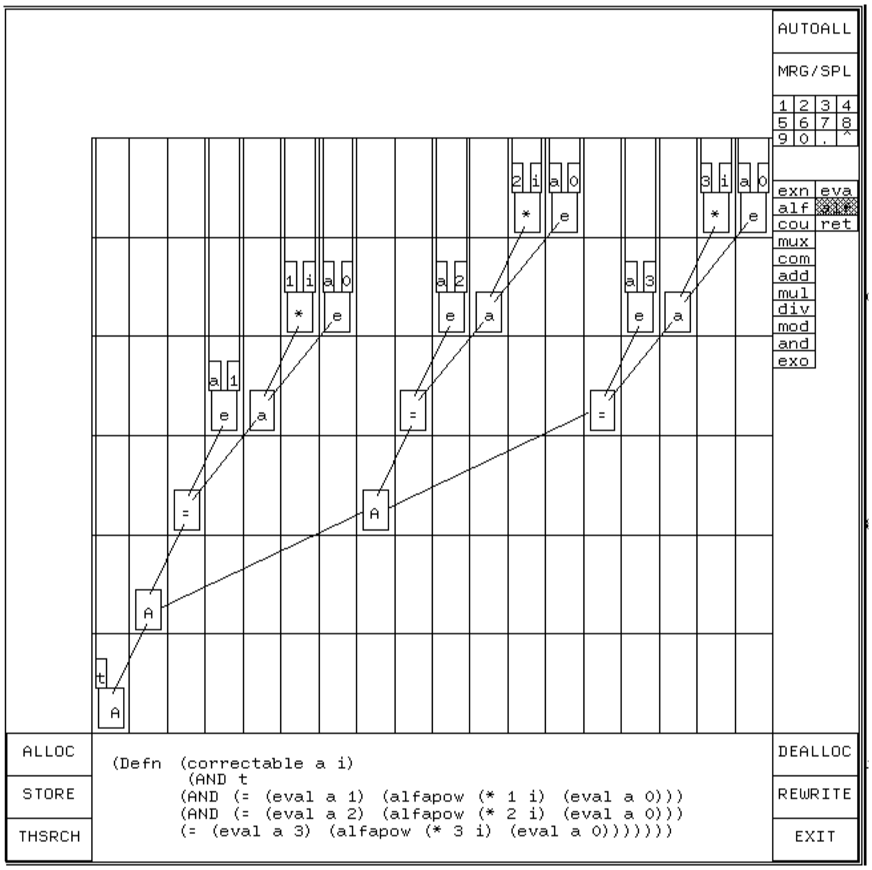

AUTOALL
MRG/SPL
1 2 3 4
5 6 7 8
9 0 . ^
exn eva
alf
cou ret
mux
com
add
mul
div
mod
and
exo
ALLOC
STORE
THSRCH
(Defn (correctable a i)
(AND t
(AND (= (eval a 1) (alfapow (* 1 i) (eval a 0)))
(AND (= (eval a 2) (alfapow (* 2 i) (eval a 0)))
(= (eval a 3) (alfapow (* 3 i) (eval a 0)))))))
DEALLOC
REWRITE
EXIT

Fig. 8

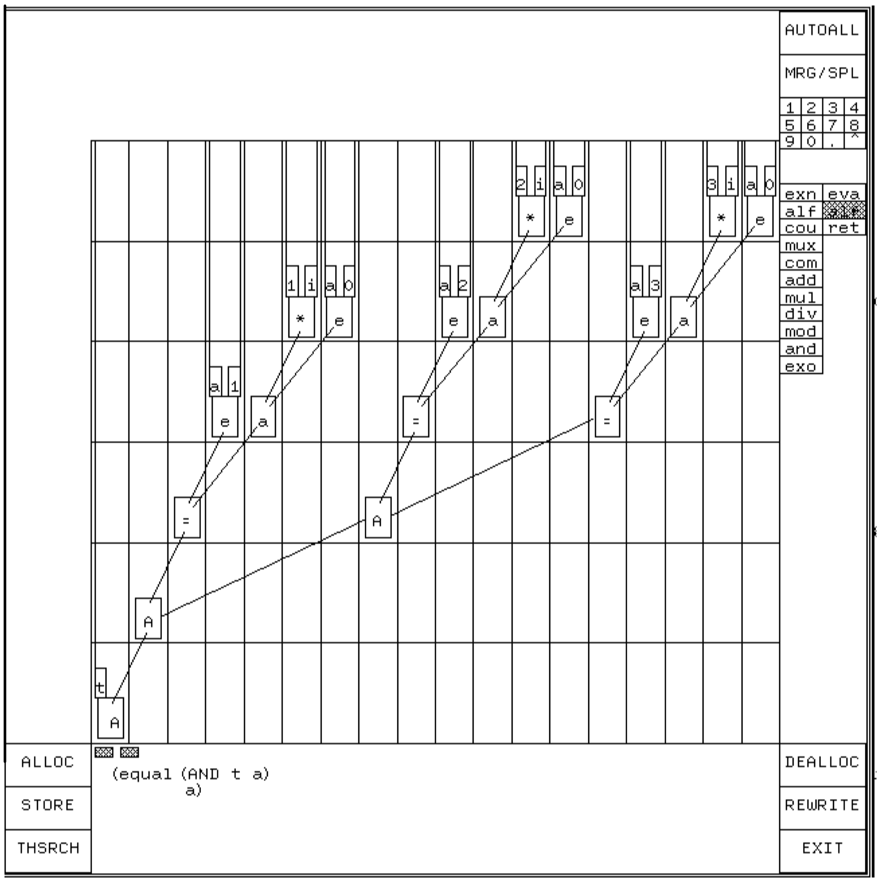

AUTOALL
MRG/SPL
1 2 3 4
5 6 7 8
9 0 . ^
exn eva
alf
cou ret
mux
com
add
mul
div
mod
and
exo
ALLOC
STORE
THSRCH
(equal (AND t a)
a)
DEALLOC
REWRITE
EXIT

Fig. 9

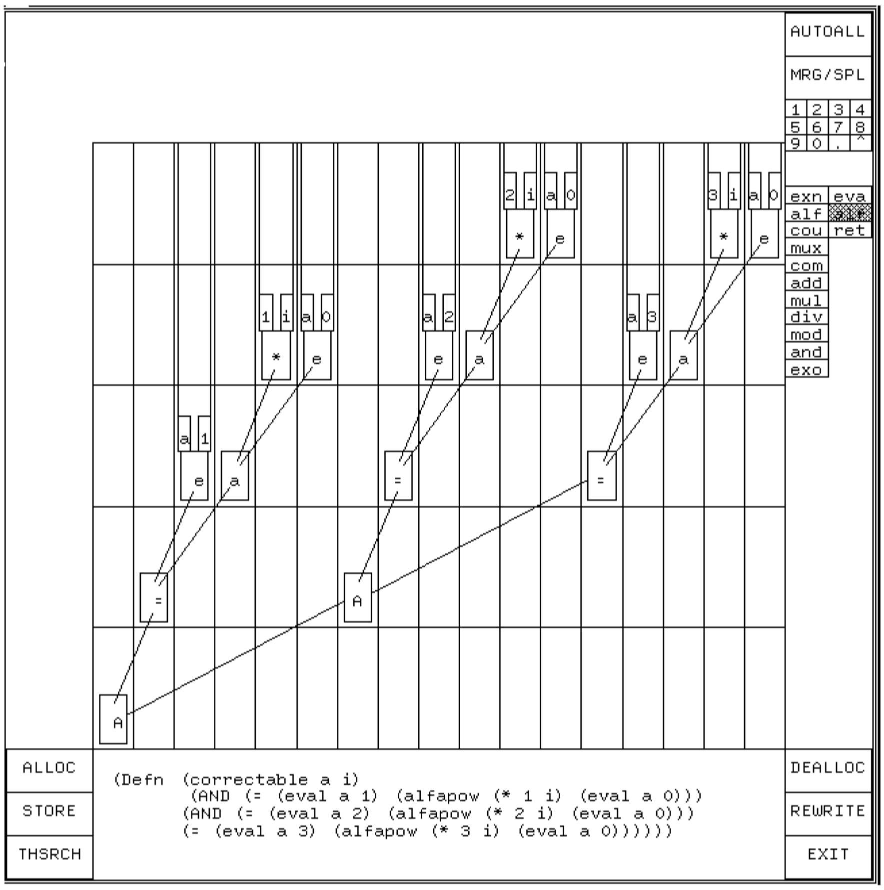

AUTOALL
MRG/SPL
1 2 3 4
5 6 7 8
9 0 . ^
exn eva
alf
cou ret
mux
com
add
mul
div
mod
and
exo
ALLOC
STORE
THSRCH
(Defn (correctable a i)
(AND (= (eval a 1) (alfapow (* 1 i) (eval a 0)))
(AND (= (eval a 2) (alfapow (* 2 i) (eval a 0)))
(= (eval a 3) (alfapow (* 3 i) (eval a 0))))))
DEALLOC
REWRITE
EXIT

Fig. 10

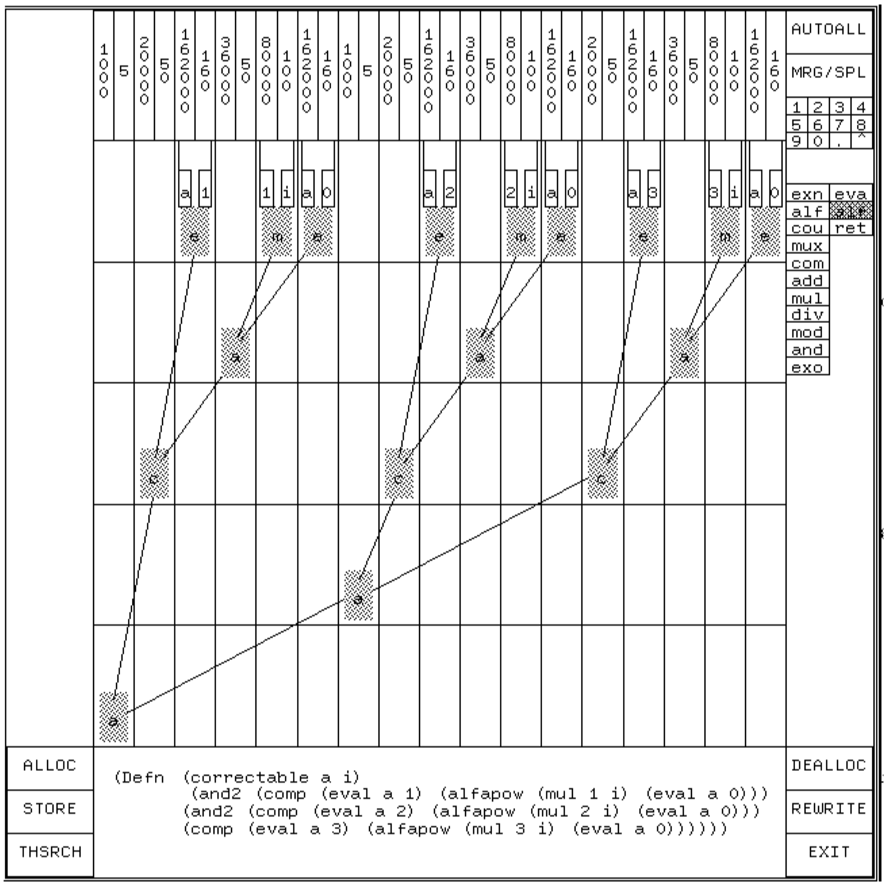
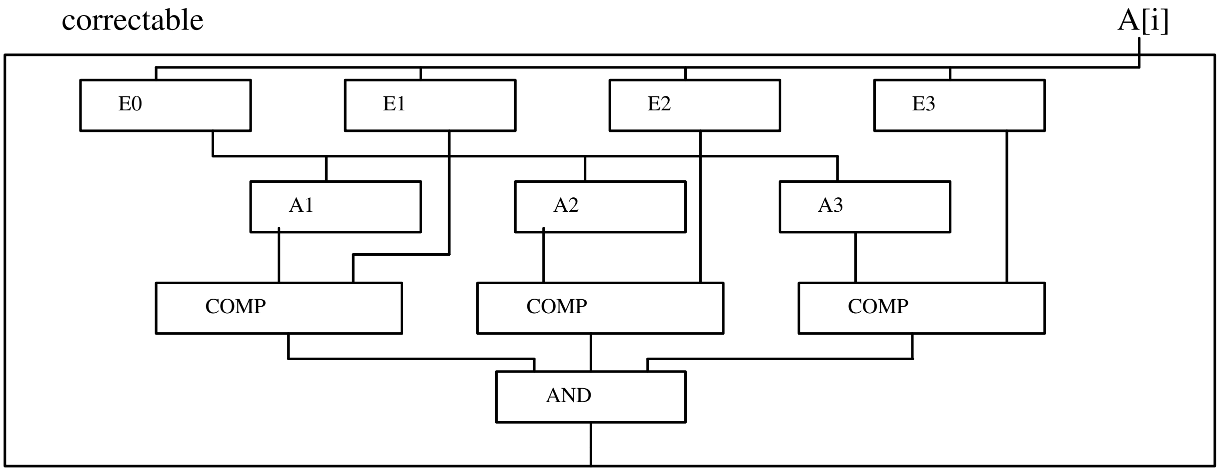



Fig. 11

A block diagram of the resulting implementation is shown in Fig. 12. We have abbreviated eval(a,0) to E0 etc., and alfapow(1*i, eval(a, 0)) to A1 etc. Upon exit, the tool will generate a complete structural STRICT description of the design, which can subsequently be used to generate its final layout.




Block diagram of implementation

Fig. 12

# 7 Conclusions

We have presented a prototype high level synthesis tool that integrates a theorem prover. The tool enables the designer to move towards an intended goal quickly and efficiently. The formal approach prevents a large class of human design errors and takes away a good deal of unnecessary effort on the part of the designer, as well as eliminating the need for much of the simulation now used to verify designs before fabrication. Whilst we have shown that a theorem prover can provide considerable benefits in the design of hardware, there are some areas where further work should be done to gain the maximum advantage from the use of formal methods. These include the following:

At present the prover has to be run separately from the tool. This situation is difficult to avoid because it runs under an interpreted Lisp environment. We are investigating how to couple them more closely.

The handling of the lemma libraries is very basic. A more sophisticated library interface would increase the ease with which the tool can be applied to real designs.

Timing aspects of a design are not described formally – possible changes are only functional, and the transformations do not take into account constraints that need to be adhered to regarding such timings. Correct combined time and space manipulations would also be useful.

## 8 References


[Bo88] Boyer, R.S., Moore, J.S. *A Computational Logic Handbook*. Academic Press, 1988.

[Br88] Brayton, R.K., Camposano, R., DeMicheli, G., Otten, R.H, vanEijndhoven, J. *The Yorktown Silicon Compiler.* In: "Silicon Compilation", Gajski, (Ed), Addison Wesley, 1988.

[Ca84] Campbell, R.H., Koelmans, A.M., McLauchlan, M.R. *STRICT : a design language for strongly typed recursive integrated circuits.* IEE Proc. Vol 132, Pts E and I, No 2, 1985.

[Ca88] Camposano, R. *Behavior–preserving transformations for High Level Synthesis*, Lecture Notes in Computer Science 408, Springer Verlag, pp. 106–128.

[De86] DeMan, H., Rabaey, J., Six, P., Claesen, L. *Cathedral II: A Silicon Compiler for Digital Signal Processing.* IEEE Design and Test, Vol 3, No 6, Dec. 1986.

[De89] Denyer, B.P. *SAGE, A Methodology and Toolset for Architectural Synthesis.* Technical Report, Department of Electrical Engineering, Edinburgh University, 1989.

[El89] Elmasry, M.I., Buset, O.A. ACE, *A Hierarchical Graphical Interface for Architectural Synthesis.* Proc. 26th ACM/IEEE Design Aut. Conf., 1989.

[Fi89] Finn, M., Fourman, M.P., Francis, M., Harris, R. *Formal System Design – Interactive synthesis based on Computer–assisted Formal Reasoning.* Proc. IFIP workshop "Applied Formal Methods for Correct VLSI Design", Belgium, 1989.

[Go87] Gordon, M. *HOL: A Proof Generating System for Higher Order Logic.* In: "VLSI Specification, verification and systhesis", G. Birtwistle, P.A. Subramanyam (Eds), Kluwer, 1987.

[Hu85] Hunt, W. *FM8501: A Verified Microprocessor.* PhD Thesis, University of Texas At Austin, 1985.

[IE87] *IEEE Standard VHDL Language Reference Manual*, IEEE Press, 1987.

[Ka88]. Kalker, T. *HOL Semantics for DSP*, Philips Research Labs Technical Report, Eindhoven, The Netherlands. 1988.

[Mc88] McFarland, S.J., Parker, A.C., Camposano, R. *Tutorial on High Level Synthesis.* Proc. 25th ACM/IEEE Design Aut. Conf., IEEE, 1988.

[Pi89a] Pierre, L. *The formal proof of the "min–max" sequential benchmark described in CASCADE using the Boyer–Moore theorem prover.* Proc. IFIP workshop "Applied Formal Methods for Correct VLSI Design", Belgium, 1989.

[Pi89b] Pierre, L. *The formal proof of sequential circuits described in CASCADE using the Boyer–Moore Theorem Prover.* Proc. IFIP workshop "Applied Formal Methods for Correct VLSI Design", Belgium, 1989.

[Ve88] Verkest, D., Johannes, P., Claesen, L., De Man, H. *Formal Techniques for Proving Correctness of Parameterized Hardware using Correctness Preserving Transformations.* In Proc. IFIP workshop "The Fusion of Hardware Design and Verification". North Holland, 1988.