Correct Interactive Transformational Synthesis of DSP Hardware

F.P. Burns, D.J. Kinniment, A.M. Koelmans
Correct Interactive Transformational Synthesis of DSP Hardware

F.P. Burns, D.J. Kinniment, A.M. Koelmans

Abstract

We present a new interactive tool for the guided synthesis of digital signal processing hardware. The tool is driven from a HDL. It will suggest different ways of implementing different architectures for the same specification, maintaining the correctness of implementations during the design process. The tool will automatically generate input for the Boyer Moore theorem prover from the HDL specification in order to verify the correctness of the implementations.


©1991 University of Newcastle upon Tyne.
Printed and published by the University of Newcastle upon Tyne, Computing Laboratory, Claremont Tower, Claremont Road, Newcastle upon Tyne, NE1 7RU, England.
Bibliographical details

BURNS, Frank P.

Correct interactive transformational synthesis of DSP hardware. [By] F.P. Burns, D.J. Kinniment, A.M. Koelmans

Newcastle upon Tyne: University of Newcastle upon Tyne: Computing Laboratory, 1991.

(University of Newcastle upon Tyne, Computing Laboratory, Technical Report Series, no. 321)

Added entries

KINNIMENT, David John
KOELMANS, Albertus Maria
UNIVERSITY OF NEWCASTLE UPON TYNE.

Abstract

We present a new interactive tool for the guided synthesis of digital signal processing hardware. The tool is driven from a HDL. It will suggest different ways of implementing different architectures for the same specification, maintaining the correctness of implementations during the design process. The tool will automatically generate input for the Boyer Moore theorem prover from the HDL specification in order to verify the correctness of the implementations.

About the authors

Mr. F.P. Burns is a Ph.D. student in the Department of Electrical and Electronic Engineering.
Professor D.J. Kinniment is a professor in the Department of Electrical and Electronic Engineering.
Mr. A.M. Koelmans joined the Computing Laboratory as Lecturer in March, 1985, after two years as Research Associate on the VLSI project in the Department of Electrical and Electronic Engineering.

Suggested keywords

FORMAL SPECIFICATION LANGUAGES
GUIDED SYNTHESIS
THEOREM PROVING FOR DESIGN CORRECTNESS
TRANSFORMATIONAL DESIGN AND VERIFICATION

Suggested classmarks (primary classmark underlined)
Dewey (18th): 001.6424 001.6425 511.3
U.D.C.: 519.682 681.322.06 510.64
CORRECT INTERACTIVE TRANSFORMATIONAL SYNTHESIS OF DSP HARDWARE

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

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

Abstract

We present a new interactive tool for the guided synthesis of digital signal processing hardware. The tool is driven from a HDL. It will suggest different ways of implementing different architectures for the same specification, maintaining the correctness of implementations during the design process. The tool will automatically generate input for the Boyer Moore theorem prover from the HDL specification in order to verify the correctness of the implementations.

Keywords

Transformational design and verification, guided synthesis, theorem proving for design correctness, formal specification languages.

1 Introduction

Currently, many researchers are working on the development of high level synthesis systems. State of the art examples of such systems are described in [De86,Br88]. They all need to perform a number of tasks [Mc88]. They start with a user specified behavioural description of the design. This description is transformed into an internal format, which is subsequently used for scheduling and allocation. During this phase 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. Finally, the actual chip layout is produced.

There appear to be two main strategies for handling the scheduling/allocation phase: fully automatic [De86,Br88] and interactive. A recent example of an interactive tool is SAGE [De89]. This tool allows various tradeoffs between area and time to be explored interactively by the designer, by, for instance, serialising operations upon the same device or by allocating many operations to devices operating in parallel. This approach encourages designers to investigate different designs through experimenting with different architectures generated from the same specification. In contrast, fully automatic synthesis systems rely upon sophisticated design space exploration algorithms to generate a reasonably optimal circuit layout given the hardware constraints.

Whichever method is being used, the initial specification is likely to be inefficient, so that it will be necessary to perform transformations which preserve the intended behaviour of the design, but generate a much more efficient layout [Mc88]. The problem here is that of verification: to prove 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. Work in this area is reported in [Ve88, El89,Fi89]; each of these tools performs part of the verification effort in various ways.
This paper describes an interactive synthesis system which takes a different approach to the verification problem, both of the entire design and of applied behavioural transformations: it 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 and is interactive, and its design was inspired by the SAGE tool. The aim is to make it easier for the designer to ensure correctness of the final implementation without losing the engineering insight into design tradeoffs available from human interaction.

The paper is organized as follows. In section two we describe the basic operation of the tool. In section three, basic theory about manipulation of the functional tree is described. Section four presents a worked out example.

2 Method

specification

The behavioural specification for our system is written initially in the STRICT language [Ca85], a hardware description language in the same class as VHDL [IE87]. In STRICT, the behavioural specification for each block is mandatory, and is captured in a functional language. Our tool transforms the specification into an intermediate format which represents the corresponding functional tree. This tree is drawn on a screen by a graphical tool with which the designer can subsequently interact by selecting particular nodes and performing various operations on them, such as performing space vs. time tradeoffs or hardware allocation. 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 are verified by a theorem proving tool, the Boyer Moore theorem prover [Bo88], and its extensions to deal with sequential systems [Pi89a, b]. The Boyer Moore theorem prover was chosen for this purpose because it is a tool which requires minimum user interaction to perform its task, compared to other such tools. It is also readily available.

transformations

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 we use the Boyer Moore theorem prover for the purpose of verification, the intermediate code we have chosen to use is the S-expression based format that is used with the theorem prover. Theorems are first verified using the theorem prover, and are then applied to the functional tree, thus producing a correct modification. The theorem prover ensures that the changes made are carried out within a formal framework. Once the designer is satisfied with the various transformations applied, hardware can then be allocated.

hardware allocation

The allocation of hardware is split into two main parts, the first of which is concerned with the functional aspects of the specification. The designer can choose basic hardware modules (such as basic logic gates, adders etc) each of which has a behavioural description written in a functional language.

The second area of hardware allocation steers away from the functional aspects of the specification and is concerned with memory allocation, or the allocation of storage and communication and control between the basic hardware modules. This particular part of the implementation is performed automatically and little or no interaction is required on the part of the designer.

layout generation

Once the transformation and hardware allocation has been completed, a STRICT description of the complete design is generated, and a complete layout can then be generated from the STRICT description using standard floorplanning and routing tools.

3 Tree Manipulation

The work described here concentrates mainly on the initial phases of the design cycle as described in section 2. The tool takes the specification from the STRICT description, and once this has been converted into the
functional tree, its representation is shown on the screen, and the option of transforming or optimising it is open to the designer. Fig. 1 shows a picture of a typical screen.

![Figure 1](image)

fig 1

To apply a change to a particular point, a node from the tree is selected to which the change can be made. This change will be in the form of a rewrite-rule. The left hand side of the rewrite-rule is matched against the section of the tree from the chosen node upward, and the tree structure at this point is checked as well as the types of all the parameters. If the match is valid the change will be applicable. The right hand side of the rewrite-rule containing the new structure is substituted in place of the old structure and all connecting nodes appended to the new section of tree.

The tree structure of any rewrite-rule is checked against the corresponding tree structure belonging to a node in the tree, and if there is a type mismatch or the tree structures are incompatible the rewrite-rule is dismissed by the system as being incompatible. This ensures that any changes made by the user to the original behavioural description are always correct.

library organisation

To apply the changes to the functional tree we need a library of potential rewrite-rules which can be selected quickly and applied to the nodes in the tree. The nodes in the tree are used to determine which rewrite-rules can be applied. The types shown in fig. 2, including primitives and a function which represents a subtree, are available:

- TIMES
- ADD
- MINUS
- DIVIDE
- GREATER
- IF
- CONDITION
- EQUALITY
- LESS
- NOT
- AND
- OR
- FUNCTION

fig 2

The library is arranged in sections where each section corresponds to one of the nodes shown in Fig. 2. If, for example, the designer indicates that he would like to change an 'add' node on the functional tree, then the rewrite-rules related to the add node are made available. This can be done by selecting the appropriate node in the tree; the system responds by writing the available rewrite-rules from the library to the bottom section of the screen.

The rewrite-rules are represented textually as S-expressions. For example, the rewrite-rule representing the associativity of the AND operator would be

```
(equal (and a b) (and b a))
```
In addition, rewrite-rules can be applied to integers and booleans in the Boyer Moore logic; by using the shell principle[Hu85], rewrite-rules can be written to act on bit vectors as well.

**modifications to the tree**

Some of the rewrite-rules belonging to the IF node are as follows:

1/ (equal (ifr (equal a b)
        c d)
     (ifr (not (equal(a b))
         d c)))

2/ (equal (ifr (or (lessp a b)
        (equal a b))
     c d)
     (ifr (greaterp a b)
         d c))

3/ (equal (ifr a b c)
     (ifr (not a)
         c b))

4/ (equal (ifr (or (greaterp a b)
        (equal a b))
        c d)
     (ifr (lessp a b)
         d c))

where the definition of 'ifr' is

(Defn ifr (a b c)
     (if a b c)
)

The parameters a, b, c and d are all natural numbers. They can also be functions providing they return a value of type natural number.

If the designer wishes to do a rewrite-rule search to find the rules applicable to a function belonging to the main design tree, such as

(if (or (lessp w x) (equal w x)) y z)

then he has to select the IF node, and click on the rewrite-rule search icon. The system will respond by looking through the above list, and it will determine that rules 2 and 3 are applicable, and present them to the designer as options.

There are a number of different types of design which will require different types of changes. Some of these might be complicated designs which can be modified by making individual changes at specific points like the example in section 4. Other designs may be of a regular nature and have a repetitive structure which can only be modified efficiently by applying a change repeatedly throughout. For this purpose a mapping function is provided which can search through a tree from a specified point and apply the changes globally.

**recording the development**

The designer may at some time like to make a change to the functional tree using a rewrite-rule which is not already contained within the library of rewrite-rules. Such a rule might be user generated. The tool will allow the option of using such a rule. Initially the entered rewrite-rules will be stored in a file separate from the main library of rewrite-rules. It is the user's own decision to ensure that anything he inserts in the tool as a rewrite-rule is valid and to ensure that the new rules are checked before they are added to the main library.

Changes made to the original functional tree by the designer are recorded by the tool. These changes are stored in a file for future reference. This is deemed to be 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 generates 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 using previous ones. If such a rewrite-rule is stored, a subsequent implementation may be achieved more quickly by applying the new rule to the specification.

**hardware allocation**

Once the design tree is in an acceptable form, actual hardware will be allocated to the various functions. 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 made available so that the designer can choose from a variety of selections. Area and time information is shown in graphical form as he chooses his components and allocates them.

A problem that arises during hardware allocation is that the library modules have a behavioural function which can be expressed in many different ways. To allocate specific hardware, the behaviour of a component from the library has to be matched against the behaviour of the corresponding node in the functional tree. If the two do not match, the designer will have to modify the functional tree by use of the rewrite-rules until a match is obtainable.

For example, suppose we have as part of the main tree a subtree with the behaviour

IF \( b < a \) THEN \( a \) ELSE \( b \)

and we have a component in the hardware library with the behaviour

IF \( a > b \) THEN \( a \) ELSE \( b \)

then the rewrite-rule \( b < a = a > b \) will have to be selected from the library and applied to the node with the behaviour \( <(b, a) \) in order to be able to move towards a match. Application of the rewrite-rule will result in the desired behaviour.

4 Example

As an example we have taken a look at a typical non-trivial signal processing example: an error correcting function for a decoder. The decoder is taken from [Ka88]. Briefly, the function takes as its input a 32 byte vector, performs an error check on it, and will correct a specified set of disturbances if found. The **strict** behaviour for the example is shown below.

**WHENEVER**

\( \text{select1} = 1 \):

**WITHIN** \( t1 \):

**SET**

\( i = (i + 1) \mod 32 \)

\( a[i] = \text{input} \);

\( \text{select2} = 1 \):

**WITHIN** \( t2 \):

**SET**

\( i = (i + 1) \mod 32 \)

\( \text{output} = a[i] + \text{error}(a[i], i) \);

**WHERE**

\( \text{error}(a[i]): \text{byte}, i:\text{integer}) : \text{byte} := \)

**IF** correctable\( (a[i], i) \) **THEN** eval\( (a, 0) \)

**ELSE** 0

\( \text{correctable}(a: \text{byte}[32], i:\text{integer}) : \text{BOOLEAN} ::= \)

\( (\text{eval}(a, 0) = \text{alfapow}(0*i, \text{eval}(a, 0))) \text{ AND } (\text{eval}(a, 1) = \text{alfapow}(1*i, \text{eval}(a, 0))) \text{ AND } \)
(eval(a, 2) == alfaxpow(2*i, eval(a, 0))) AND
(eval(a, 3) == alfaxpow(3*i, eval(a, 0)))

eval(a: byte[32], j: integer):byte :=
    sum(31, j, a)

sum(i: integer, j: integer, a: byte[32]):byte :=
    IF i = 1 THEN partsum(a, i, j)
    ELSE exor(partsum(a, i, j), sum(i-1, j, a))

partsum(a: byte[32], i, j: integer):byte :=
    alfaxpow(j*i, a[i])

alfapow(n: integer, g: byte):byte :=
    IF n = 0 THEN g
    ELSE alfaxpow(n-1, alfax(g, 1, 8))

alfax(g: byte, i, j: integer):byte :=
    IF i < 6 AND i > 2
    THEN (exfn(g[j], g[i-1]) + 2*(alfax(g, i+1, j)))
    ELSE IF i = 1 THEN (g[j] + 2*(alfax(g, i+1, j)))
    ELSE IF i = j THEN g[i-1]
    ELSE g[i-1] + 2*(alfax(g, i+1, j))

exor(g, a: byte, i: integer):byte :=
    IF i = 8 THEN exfn(g[i], a(i))
    ELSE exfn(g[i], a[i]) + 2*(exor(g, a, i-1))

exfn(a,b: bit): bit :=
    ((~a) AND b) OR ((~b) AND a)

The specification suggests that when the function is activated the input is stored in a specific array variable a[i]. When output is required, it is set to the specific array variable a[i] with an error correction determined by the function 'error'. If we were to implement the error correcting function exactly in this fashion, it would be highly inefficient. The functional tree which follows the specification above is shown in fig. 3.

![Functional Tree](image)

Any modifications that might be made on this tree rely upon the use of rewrite–rules from the library, such as the following:

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)))
5 / (equal (alfapow (times n i) g)
     (alfapow (times 1 i)
     (alfapow (times (sub 1 n) i) g)))
6 / (equal (and (equal A (alfapow n C))
     (equal B C)
     (equal (alfapow n B))
     (equal (alfapow n B) (alfapow n C))))
7 / (equal (sum i j a) (newsum i j a))

where the definition of sum and newsum are:

(Defn sum (i j a)
     (if (zerop i) (alfapow (times i j) (lbit a))
     (exor (alfapow (times i j) (lbit a))
     (sum (subl i) j (lvec a))))))

(Defn newsum (i j a)
     (if (zerop i) (lbit a)
     (exor (alfapow j (lbit a))
     (alfapow j (newsum (subl i) j (lvec a))))))

fig 4

The diagram in fig. 4, shows the resulting tree after the application of rewrite-rules 1 to 3 to the upper left hand part of fig. 3. The transformations are applied as follows:

A to B: rewrite-rule 1 is applied to delete the multiplication node,

B to C: Alfapow(0,eval(a,0)) is replaced by eval(a,0)
by rewriting,

C to D: rewrite-rule 2 is applied to c0,

D: rewrite-rule 3 is applied to the AND node.

The final functional tree after the above changes have been applied is shown in fig. 5.
Rewrite-rule 5 can now be applied to the node a2 in the functional tree. Thus the right hand side of the second equation of the correctable function would be modified from
\[
\text{alfapow}(2\times i, \text{eval}(a, 0))
\]
to
\[
\text{alfapow}(1\times i, \text{alfapow}(1\times i, \text{eval}(a, 0))).
\]
Similarly, the process can be applied to node a3 on the functional tree twice. After this the specification for the correctable function would be as follows:
\[
\text{correctable}(a: \text{byte}[31], i: \text{integer}): \text{BOOLEAN} :=
\begin{align*}
& ((\text{eval}(a,1) == \text{alfapow}(1\times i, \text{eval}(a, 0))) \text{ AND } \\
& ((\text{eval}(a,2) == \text{alfapow}(1\times i, \text{alfapow}(1\times i, \text{eval}(a, 0)))) \text{ AND } \\
& ((\text{eval}(a,3) == \text{alfapow}(1\times i, \text{alfapow}(1\times i, \text{alfapow}(1\times i, \\
& \text{eval}(a, 0))))))))
\end{align*}
\]
The upper section of the functional tree after the above modifications is shown in fig. 6. Here it can be seen that the node a2 has been split into two a1 nodes and the node a3 has been split into three a1 nodes respectively.

\[\text{fig 6}\]

In what follows, rewrite-rule 6 is applied to the AND node in connection with (c1 and c2) or (c2 and c3). It is assumed that rewrite-rule 4 is applied to the AND node to toggle between the two expressions (and (c1 and c2 c3)), (and (and c1 c2) c3) in order to allow rewrite-rule 6 to be applied to the appropriate pair.

The sixth rewrite-rule can now be applied to the AND node in connection with the last two equations of the correctable function. Here, A=eval(a,3), B=eval(a,2) and C=alfapow(1\times i, alfapow(1\times i, eval(a,0))). The resulting tree is shown in fig. 7.

\[\text{fig 7}\]
The fifth equality is now applied in reverse, i.e.
\[
\text{(equal (alfapow (times n i) (alfapow (times m i) g))}
\]
\[
\text{(alfapow (times (plus n m) i) g))}
\]
to the alfapow node marked by the star in the above diagram. The effect of this is to reduce the expression
\[
\text{alfapow}(1\times i, \text{alfapow}(1\times i, \text{alfapow}(1\times i, \text{eval}(a, 0))))
\]
down to the following expression:
\[
\text{alfapow}(2\times i, \text{alfapow}(1\times i, \text{eval}(a, 0))).
\]
The corresponding expression and functional tree for the correctable function is as follows:

correctable(a: byte[31], i: integer): BOOLEAN ::= 
  (eval(a, 1) == alfpow(1*i, eval(a, 0))) AND
  (alfpow(1*i, eval(a, 2)) ==
      alfpow(2*i, alfpow(1*i, eval(a, 0)))) AND
  (eval(a, 3) == alfpow(1*i, eval(a, 2))

\[ \text{fig 8} \]

The sixth equality can be applied again but this time in connection with the first two equations of the correctable function, in this case \( A = \text{alfpow}(1*i, \text{eval}(a,2)) \), \( B = \text{eval}(a, 1) \), \( C = \text{alfpow}(1*i, \text{eval}(a, 0)) \). The modified version should end up looking like the functional tree below.

\[ \text{fig 9} \]

The inverse of the fifth equality is applied again to the node in Fig. 9 marked by the star, to produce:

correctable(a: byte[31], i: integer): BOOLEAN ::= 
  ((alfpow(2*i, eval(a, 1)) == alfpow(3*i, eval(a, 0)))) AND
  ((alfpow(1*i, eval(a, 2)) == alfpow(2*i, eval(a, 1)))) AND
  ((eval(a, 3)) == alfpow(1*i, eval(a, 2)))

\[ \text{fig 10} \]

The seventh rewrite-rule can be applied to the function sum in the specification above. First of all, partsum must be replaced with alfpow before applying the rewrite-rule (this is simply a case of substitution by rewriting). The modified version of the specification in this case would be:

\[
\text{newsum}(i: \text{integer}, j: \text{integer}, a: \text{byte}[32]):\text{byte} ::= \\
\text{IF } i = 1 \text{ THEN } a[i]
\]
ELSE \text{exor(alfapow}(j, a[i]), alfapow}(j, \text{newsum}(i-1, j, a)))

The reduction of the calls to the function alfapow is obtained by nesting newsum within its call. Thus, the number of calls to the subfunction alfax is reduced by a factor of \(i\). The diagrams in fig.11 and fig.12. show the difference in implementation resulting from the change. The block diagrams in fig. 13 show the implementation of the error correcting function after hardware allocation.

The modified version of newsum applies the function alfapow \(j\) times for each recursive call to newsum. Thus, in calculating the values \text{eval}(a, j)\), where \(j\) varies from 0 to 3, the number of times that the alfax function is applied varies from 0 to 3. In the case where \(j>1\) we have the choice of serialising the operations on one device or of expanding the hardware. Fig. 14 shows the expanded version for the case where \(j=3\).
5 Conclusions

The work described in this paper is presently at an advanced stage. 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.

The tool has some limitations that could be improved. For example, the 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. Also, the modules to allocate hardware are not yet integrated into the tool. In addition, correct combined time and space manipulations would also be useful. Lack of these facilities do not prevent useful application of the technique and we are investigating how they may be added.

6 Acknowledgements

The authors would like to thank Martin McLauchlan for constructive comments on an earlier draft of the paper.

7 References


