Architectural Transformations for Hierarchical Algorithmic Descriptions

Marcio Yukio Teruya, Marius Strum and Wang Jiang Chau
Department of Electronic Enginnering, Escola Politecnica da Universidade de Sao Paulo, Brazil

Key words: Transformational Design, Architectural Synthesis, High Level Synthesis, System Synthesis

Abstract: The use of hierarchy on writing algorithmic descriptions of digital systems allows the implementation of more complex designs since it increases designer's productivity by introducing important features such as modularity, encapsulation and reusability. We are particularly interested in the problem of generating an optimal register transfer logic structure from a hierarchical algorithmic description. It is relatively straightforward to use High Level Synthesis (HLS) tools for producing an implementation from hierarchical algorithmic descriptions; each algorithmic partition is implemented separately and then linked in a following step. In general, the results are sub-optimal due to the large gap existing between the specification and implementation. In this article, we detail a simple architectural model for hierarchical algorithmic descriptions and a set of architectural transformations, which are the core of a methodology, Recursive High Level Synthesis (RHLS), aimed to optimise hierarchical implementations. The transformations are used to reshape the architecture of pre-existing hierarchical algorithmic descriptions in order to provide better synthesis results from HLS. We have implemented a suitable data structure and a set of transformations and tested them over a set of hierarchical algorithmic examples.

The original version of this chapter was revised: The copyright line was incorrect. This has been corrected. The Erratum to this chapter is available at DOI: 10.1007/978-0-387-35498-9_57

L. M. Silveira et al. (eds.), *VLSI: Systems on a Chip*
© IFIP International Federation for Information Processing 2000

1. INTRODUCTION

To cope with growing digital system designs complexity, a well established strategy is to represent them hierarchically. It is well known that hierarchy allows introducing some very desirable features into the design process, such as modularity, encapsulation and reusability, contributing towards increased productivity and making large projects less expensive. Although designers may have their task of specifying and describing a circuit made easier, the results at implementation side may not be as positive. For instance, the hierarchy of a design entry, containing a partitioning that aims at productivity, may show to be inefficient if directly reproduced into the implementation, whose partitioning should be driven by implementation level metrics (timing, power, cost, etc.).

In present methodologies based on higher level descriptions, the design entry presents a widening gap to the final implementation description what increases their aforementioned hierarchical incompatibility. This is very much true for algorithmic level design. Figure 1a shows an example of a hierarchical algorithmic description; **X** is the top level algorithm, **Y** and **Z** are the lower level algorithms that implement operations **opY** and **opZ**, respectively, in algorithm **X**. To produce a structural level implementation, we can use High Level Synthesis (**HLS**) [1] as shown in Figure 1b, where the synthesised structures from algorithms **Y** and **Z** are represented as blocks composed of smaller basic blocks (**A**, **B**, **C** and **D**). Figure 1c shows the synthesised structure from algorithm **X**; it preserves the original algorithmic hierarchy but it presents some redundancy; for instance, the basic block **C** appears 3 times, once at each partition. Depending on the circuit timing requirements, there could be some additional block sharing through a different partitioning, as seen in figure 2.

This example shows that, for the sake of implementation quality, the hierarchical decomposition at design entry level must be in tune with the respective one at implementation level, but, usually, when high-level partitioning is carried out, implementation issues are not visible yet. The problem here is how to keep the mentioned productivity advantages of adopting hierarchical methodologies at design entry level and still deliver efficient implementations. Some attempts have been made to deal with this problem but there is no clear solution yet. In [2][3][4], the problem of generating hierarchical structures was tackled, but the authors did not focus on productivity issues (as possible component reusability) arisen from a hierarchical strategy. Even though, their approaches were capable of producing hierarchical structures containing some optimisation, via restricted rules in partitioning of data-flow [2] or control-flow [3][4] graphs derived from plain algorithmic descriptions. In [5], it is shown a structured

methodology capable of dealing with hierarchical algorithmic designs using a HLS system, but they did not show any automatic means for producing optimised implementations. More recently, in [6], the authors presented an extended HLS system, which can allocate components covering different levels of hierarchy to the algorithmic level descriptions. They have proposed an extended model for re-configurable functional units (**FUs**) which includes their behavioural information; at the scheduling phase redundancies are detected and optimised hierarchical structures are produced. The drawback of this approach is its complexity - a whole new set of efficient algorithms must be derived, besides the fact that new generated FUs are problem dependent what restricts their reuse.

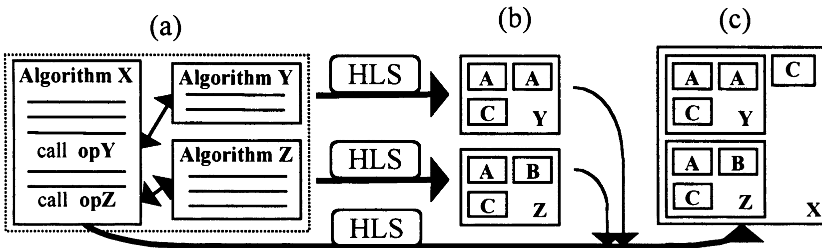

Figure 1. From a hierarchical algorithmic description to a hierarchical structure using HLS.

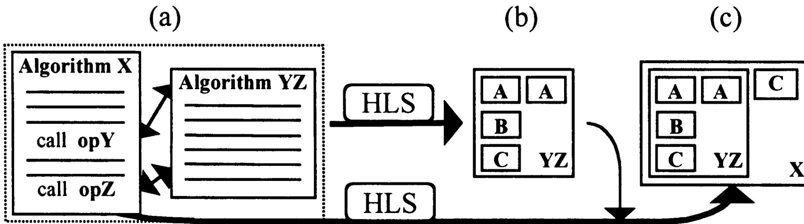

Figure 2. An alternative partitioning for the hierarchical algorithmic description.

In this paper, we follow the **Recursive High Level Synthesis (RHLS)** approach, presented elsewhere[7], to solve the problem of generating an efficient hierarchical register transfer logic (RTL) structure from a hierarchical algorithmic description. It is based on implementation metrics and transformations over hierarchical algorithmic descriptions very much like going from the one in figure 1 to the one in figure 2 - the newly created hierarchical algorithm should then be suitable for generating, via HLS, an optimised structure.

The objective of this article is to present details about the foundations of our methodology for RHLS, which are: an **architectural model for hierarchical algorithmic descriptions** and a set of **Architectural Transformations (ATs)**. Our architectural model establishes a basic framework for designing hierarchical algorithmic descriptions and also

establishes a basic vocabulary upon which the transformations will act on. It was designed to be independent of hardware description languages such as the VHDL syntax. The architectural transformations provide new modules which may fulfil better a design implementation level requirements and, since the transformations are defined on modules operation sets, they make a strong contribution for FU reuse environments. It should be noted that the concept of architectural transformations is not new; in [8], the authors presented an implementation based on an architectural model at RTL level. We have adopted a higher abstraction level architecture for two reasons: higher flexibility and simpler implementation. The set of transformations acts on the hierarchical algorithmic descriptions following our model, reshaping them for structural level optimisations, particularly by improving sharing of structural resources, such as FUs, registers and so on, supported by a HLS system, as exemplified previously.

This article is organised as follows. Section 2 gives a general explanation about our architectural model and the transformations. The sections 3 and 4 present details, respectively, of a mathematical formalism for our architecture model and the algorithms we implemented for the set of transformations. In section 5 implementation issues are explained and some results are presented. Finally, section 6 concludes this article.

2. STRATEGY OVERVIEW AND DEFINITIONS

2.1 Recursive High-Level Synthesis

The Recursive High Level Synthesis may be defined as the optimisation process of a hierarchical RTL structure through the application of a sequence of transformations on its functional units (FUs). We say *hierarchical* RTL structure because it contains FUs which are RTL structures themselves. Furthermore, these FUs also have behavioural descriptions from which a RTL structure may be obtained via HLS.

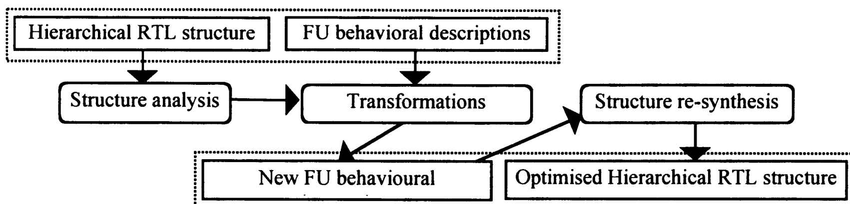

Figure 3. Tasks in RHLS

Figure 3 illustrates the main tasks in RHLS: **Structure analysis** - the hierarchical RTL structure is examined in regard to redundancies or any other inefficiencies through their functional units; **Transformations** - new functional units are created based on existing ones in order to solve problems detected by the structure analysis; and **Structure re-synthesis** - a new hierarchical RTL structure is generated using the newly created functional units. More details about RHLS can be found in [7].

2.2 Architectural Model

Our architectural model establishes the basic framework for writing hierarchical algorithmic descriptions. In other words, it defines the basic components and their basic interrelations for building hierarchical algorithmic descriptions. Therefore, besides providing guidance for writing these descriptions, it states what the transformations can manipulate.

The basic elements of our architectural model are the **behavioural views** (algorithmic descriptions) of **complex functional units**. We understand **functional unit (FU)** as a HLS system library component that is capable of performing one or more operations. **Complex FU** is a FU that holds an algorithmic description for the operation set it is capable of performing. Inside this algorithmic description, there are calls to operations provided by either primitive (non-complex) FUs or other complex FUs. This establishes a relation of hierarchy among complex FUs and, therefore, a relation of hierarchy among algorithmic descriptions. Hence, our architectural model is basically a set of intercommunicating (via operation calls) processes (algorithmic descriptions) hierarchically organised.

```
entity FU_name is
    port ( port declarations );  end FU_name;
architecture behavior of FU_name is begin process
    variable declarations
begin
    case sel is
        when 1 => (instruction set for operation 1)
        ...
        when n => (instruction set for operation n)
    end case;
end process;  end behavior;
```

Figure 4. Behavioural view of a FU represented as a VHDL template.

The architectural model also comprises the architecture of the behavioural views of complex FUs; we have modelled it as a set of **ports, variables, instruction sets, internal operations** and a **list of FUs** (figure 4). The **instruction sets** are the algorithmic descriptions of the operations the FU is

capable of performing and there is exactly one instruction set for each operation of the FU. The **internal operations** are the operations used inside the algorithmic description of the FU's behavioural view. The **list of FUs** (not shown in figure 4, but considered part of the behavioural view) refers to the *candidate* FUs to execute the internal operations.

The processes (i.e. the behavioural views of the FUs) are organised in a hierarchical fashion, therefore, the activation of any sibling process or their data exchange must be arbitrated by the parent processes using some kind of communication protocol. Our approach to solve this problem is similar to the one proposed in [5], where intercommunicating protocol is implemented at the parent processes as reproduced in figure 5. The protocol is composed of two operations: the first one (*opX*) starts the operation and sends the related parameters to the sibling process and the second one (*waitX*) runs repeatedly until the result is ready to be used. This protocol is particularly useful in processes whose processing time is data dependent.

```
opX(a,b);                    -- starts the operation with parameters a and b
waitX (c,done);              -- reads the flag (done) for operation completion
while (done /= '1') loop     -- loops until operation completes
    waitX (c,done);          -- and reads a result (c)
end loop;
```

Figure 5. A interprocess protocol in VHDL

2.3 Architectural Transformations (ATs)

We have implemented three ATs, which, essentially, perform modifications on the architecture of complex FU behavioural views, in order to produce new complex FUs with altered operation sets. More elaborate architectural changes can be achieved through combination of these basic transformations: The transformations are:

a) **Merge**: from two pre-existent FUs, **Merge** produces the behavioural view of a FU capable of performing the same operations of the two original FUs. As an optimisation tool, **Merge** is intended to cluster FUs with high probability of resource sharing.
b) **Extract**: from one pre-existent FU, **Extract** produces the behavioural view of a FU with one operation removed from the original operation set. **Extract** is intended to reshape existing FUs removing redundancies so they can better fulfil new requirements.
c) **Promote**: from one pre-existent complex FU (which contains at least one more FU), **Promote** produces the behavioural view of a FU with one extra operation in its operation set (the extra operation is borrowed from a sibling FU). **Promote** is intended to add functionality to existing FUs so they can better fulfil new requirements.

3. MATHEMATICAL MODEL

The architecture of a FU behavioural view is defined as:

$A = <name, O, P, V, \boldsymbol{I}, O', \boldsymbol{A}>$, where:

- *name* is the name of the FU.
- O is the set of operations executed by the FU; $O=<o_1,o_2, ...,o_{nO}>$, where o_i is the ith operation; $o_i=<name>$ meaning it has a *name*.
- P is the set of ports; $P=<p_0,p_1, ...,p_{nP}>$, where p_i is the ith port; $p_i=<name,width,type>$ meaning it has a *name*, a *width* (in bits) and a *type* belonging to the set <"*control*","*input*","*output*">; p_0 is of type "*control*", the other ones must be "*input*" or "*output*".
- V is the set of variables; $V=<v_1,v_2,...,v_{nV}>$, where v_i is the ith variable; $v_i=<name,width>$ meaning it has a *name* and a *width* (in bits).
- $\boldsymbol{I}$ is the set of instruction sets; $\boldsymbol{I} = < I_1, I_2, ..., I_{nI} >$, where I_j is the jth instruction set; $I_j=<i_1,i_2,...,i_{nIj}>$, where i_k is the kth instruction; $i_k=<T,B>$, where T is the type of instruction, which can be any instruction of the adopted hardware design language's (HDL) instruction set, and B is the body of the instruction which can be a list of arguments (ports, variables, constants) or another instructions.
- O' is the set of internal operations; $O'=\{o_1',o_2' ...,o_{nO'}'\}$, where o_i' is the ith internal operation; $o_i=<name,PA>$ meaning it has a *name* and a set of parameters $PA=<pa_1, pa_2, ..., pa_{nPA}>$ where $pa_i=<name,type>$ is the ith parameter; each parameter also has a name and a type belonging to the set <"*input*","*output*">.
- $\boldsymbol{A}$ is the set of architectures (it is the list of FUs mentioned in section 2.2) capable of performing the internal operations in O'; $\boldsymbol{A} = \{A_1,A_2,...,A_{nA}\}$, where A_i is the ith architecture.

Observe that the last element of this mathematical model, $\boldsymbol{A}$ (set of architectures), makes the model suitable to represent a hierarchical algorithmic description. The mathematical model has the following assumptions: 1) FUs are capable of executing one or more operations; 2) all ports are bit vectors; 3) all input and output ports can exist in any quantity and there are no width restrictions; 4) bi-directional ports are not allowed; 5) there is only one control port which is designated to activate the FU and to select one desired operation; 6) all variables are bit vectors; 7) variables can exist in any quantity and there are no width restrictions; 8) if the FU is capable of executing more than one operation, then only one operation can be executing at a time; and 9) to each instruction set corresponds one, and only one, operation. Some of these assumptions are restrictions imposed to simplify the implementation of data structures and, as such, they could be relaxed to increase the model's generality. An example of use of the mathematical model is given below for a reciprocal division module.

Listing 1: A behavioural view in algorithmic VHDL.

```
Entity mrep is
    port (input1 : in integer;  sel : in int2bit;  outval : out integer;  outdone : out bit);
end mrep;
architecture behavior of mrep is begin process
    variable A, T, X, Q : integer;
begin
    case sel is
      when 1 => -- reciprocal operation
        T:=constexp20; X:=0; A:=input1; outdone<='0';
        while (X/=20) loop Q:=shiftleft(T); X:=X+1;  T:=Q-A;
            if (T>0 and A>0) then T:=T+1; end if; T:=qsave(Q,T);
        Q:=selectlsb(T);
      when 2 => -- waiting operation
        outval <= Q;  outdone <= '1';
end case;  end process;  end behavior;
```

Listing 2: The corresponding mathematical model for listing 1.

```
Amrep = < "mrep", Omrep, Pmrep, Vmrep, Imrep, Omrep', Amrep >
    Omrep = < "rep","wrrep" >
    Pmrep = < p_0, p_1, p_2, p_3 >, p_0 =<"sel","control",2>, p_1=<"input1","input",32>,
        p_2 =<"outval","output",32>, p_3=<"outdone","output",1>
    Vmrep = <v_1,v_2,v_3,v_4>, v_1=<"A",32>, v_2=<"T",32>, v_3=<"X",32>, v_4 =< "Q",32>
    Imrep = <I_1>
        I_1 = <I_1,I_2,I_3,I_4,I_5,I_6>,  I_2 = < I_1, I_2 >
            I_1.I_1 = <ASSIGN,<v_2,<O_3',null>>>, I_1.I_2 = <ASSIGN,<v_3,0>>,
            I_1.I_3 = <ASSIGN,<v_1,p_1>>, I_1.I_4 = <ASSIGN,<p_4,0>>,
            I_1.I_5 = <WHILE,<<NE,<v_3,20>>, <O_4',<v_4,v_2>>, <O_1',<v_3,v_3,1>>, <O_2',<v_2,v_4,v_1>>,
                    <IF, <<AND,<<GT,v_2,0>,<GT,v_1,0>>>,< O_1',<v_2,v_2,1>>>>, <O_5',<v_2,v_4,v_2>>>>,
            I_1.I_6 = < O_6',<v_4,v_2>>
            I_2.I_1 = <ASSIGN,<p_3,v_4>>, I_2.I_2 = <ASSIGN,<p_4,1>>
    Omrep' = { O_1', O_2', O_3', O_4', O_5', O_6' }
        O_1' = <"+",<pa_1,pa_2>>, O_2' = <"-",<pa_1,pa_2>>, O_3' = <"constexp20",null>
        O_4' = <"shiftleft",<pa_1,pa_2>>, O_5' = <"qsave",<pa_1,pa_2,pa_3>>, O_6' = <"selectsb",<pa_1,pa_2>>,
    Amrep = { Aalu, Ashift }
```

4. ALGORITHMS

In this section we present the main algorithms we have implemented, all based on the mathematical model just introduced.

The **Merge** AT receives two architectures as inputs: *A1=<name1,O1,P1, V1,**I1**,O1',**A1**>* and *A2=<name2,O2,P2,V2,**I2**,O2',**A2**>*, and generates a third one: *Am=<namem,Om,Pm,Vm,**Im**,Om',**Am**>*, using the following algorithm:

1. *namem* = a different name from those pre-existent
2. *Om' = O1'* U *O2'*
3. *<Om,**Im**>* = **Unite(***<O1,**I1**>,<O2,**I2**>***)**
4. *<Pm,**Im**>* = **MergePortSets(***P1,P2,**Im***)**
5. *<Vm,**Im**>* = **MergeVariableSets(***V1,V2,**Im***)**
6. ***Am* = *A1*** U ***A2***
7. Return *Am* = *<namem,Om,Pm,Vm,**Im**,Om',**Am**>*

First (line1), a name different from any other is given to the architecture, then, the element *Om'* (line 2), the set of internal operations, is generated by simple union of those in the initial architectures; the same occurs for the element ***Am*** (line 6), set of architectures. The elements *Om*, set of operations, and ***Im***, set of instruction sets, are produced by the special function **Unite** (line 3) which works similarly to the connective set union; regarding ***Im***, there is an one-to-one correspondence between the sets *O* and *I*: ***Im*** is generated taking *Om* as guide, i.e., if the first element of *Om* is the first element of *O1*, then the first element of ***Im*** must be the first element of ***I1***, and so on. Finally, the elements *Pm* and *Vm* are generated by algorithms **MergePortSets** and **MergeVariableSets** which perform four similar tasks. **MergePortSets** concatenates the original port sets, minimises them, creates a new control port with a suitable width and updates the set of instruction sets with the new ports. To minimise the port set, an algorithm aimed to share ports among instruction sets is used- given a pair of ports, one can substitute the other if they are of the same type (input or output), have the same width and belong to different instruction sets.

The **Extract** AT has the form **Extract**(*A*,*opname*). It receives one architecture as input, *A*=<*name*,*O*,*P*,*V*,***I***,*O'*,***A***>, and the name of the operation being extracted, *opname*, and it generates a second architecture: *Ae*=<*namee*,*Oe*,*Pe*,*Ve*,***Ie***,*Oe'*,***Ae***>. The undesired operation is extracted from the original operation set, *O*, to produce the new operation set *Oe*; *Ie* is generated in a similar way. The other elements of the new architecture (*Pe*, *Ve*, *Oe* and *Ae*) are derived from the original one by eliminating all the elements that became unused after removing the instruction set corresponding to the extracted operation.

The **Promote** AT has the form **Promote**(*A*,*opname*). It receives one architecture as input, *A*=<*name*,*O*,*P*,*V*,***I***,*O'*,***A***>, and the name of the operation being promoted, *opname*, and it generates a second architecture: *Ap*=<*namep*,*Op*,*Pp*,*Vp*,***Ip***,*Op'*,***Ap***>. The algorithm generates a new operation set, *Op*, appending the new operation to the pre-existing set of operations. The new port set, *Pp*, is generated by concatenating the existing port set, *P*, to a new one, *Pnew*, containing the exact number of ports required by the promoted operation. After this, a new instruction set is created for the promoted operation; it is composed of a simple call to the new operation having the new ports, *Pnew*, as parameters and the created port set, *Pp*, is minimised. A new control port is generated according to the number of operations and it is appended to *Pp*. Finally, the elements *Vp*, *Op'* and ***Ap*** are directly copied from the original architecture. It should be observed that a promoted operation requires a costlier access time compared to a identical non-promoted operation because promotion implies arbitration through FU's top controller (one extra cycle clock in our case).

5. RESULTS

The first set of results (tables 1 and 2) illustrates the AT's *modus operandi*; they were obtained from transformations over three benchmarks (*gcd*, *mmul* and *mrep*) and from HLS. Table 1 shows the architectural characteristics of the benchmarks and resulting architectures from merging, promoting and extracting. The table shows the operations the FUs are capable of performing (*O*), its ports (*P*), its variables (*V*) and its architecture set (***A***).

Table 1. Architectural characteristics for the benchmarks and the ones produced by the ATs

FU	*O*	*P*	*V*	*A*
mgcd	gcd, wgcd	2 i[32], 1 o[32], 1 o[1], sel[2]	2[32]	alu
mmul	mul, wmul	2 i[32], 1 o[32],1 o[1], sel[2]	5[32]	alubase,shift
mrep	rep, wrep	1 i[32], 1 o[32],1 o[1], sel[2]	4[32]	alu,shift
Merge(*mrep,mmul*)	mul, wmul, rep, wrep	2 i[32], 1 o[32],1 o[1], sel[3]	5[32]	alu,shift,alubase
Promote(*mgcd*,{-})	gcd,wgcd, -	2 i[32], 1 o[32],1 o[1], sel[2]	2[32]	alu
Extract(Promote (*mgcd*,{-}),{-})	gcd, wgcd	2 i[32], 1 o[32],1 o[1], sel[2]	2[32]	alu,alubase

Notes: Column *P*: i designates a input port, o, an output port and sel, a selection port; the number between brackets indicates the port's width in bits. Column *V*: the number between brackets indicates a variable with that many bits. Alu, alubase and shift are primitive FUs.

Table 2 shows characteristics of RTL structures obtained from the architectures in table 1, via a HLS system (we have used AMICAL[9]). Regarding **Merge**, it can be observed a reduction on FU counting: the structure for **Merge**(*mrep,mmul*) uses 1 *alubase* and 2 *shifts*; if *mmul* and *mrep* were taken separately, the total of FUs would be 1 *alubase*, 3 *shifts* and 1 *alu*. The same can be said about the muxes. Regarding number of registers, AMICAL allocates one register for each variable and constant, therefore it closely follows the variable set. The downside is on the controller - it is larger than the sum of the ones in the original FUs, although the number of states, transitions and ios (total sum of controller's inputs and outputs) were smaller as shown in the column 'controller' in table 2. This happened because controller area are non-linearly dependent on the number of states, transitions and ios. Adding data-path and controller, the FU produced by **Merge** is smaller than the sum of the originating ones; the column 'gain' shows the difference. Regarding **Promote**, it forces the creation of paths from ports to FUs; this effect can be observed in table 2, which shows an increased number of muxes and controller hardware when comparing transformed units to the original ones. However, using **Promote** may still be advantageous; for instance, if a data-path contains a *mmul* and an *alu*, and **Promote** is applied in *mmul* generating a new version which is capable of performing all the operations of an *alu*, then the data-path could not need the *alu* anymore. In table 2, the column 'gain' shows the gain in area if the aforementioned optimisation were carried out.

Table 2. Structural characteristics for the benchmarks and for the ones produced by the ATs.

	structural features	area	gain
mgcd	1 FU (alu), 4 Regs, 5 Muxes, (6T,2S,145ios)	4744+604=5348	-
mmul	3 FUs (alubase, shift(2)), 7 Regs, 8 Muxes, (12T,6S,160ios)	12008+1346=13354	-
mrep	2 FUs (shift, alu), 6 Regs, 6 Muxes, (13T,7S,215ios)	8680+1952=10632	-
Merge(*mrep*, *mmul*)	3 FUs (alubase,shift(2)), 7 Regs, 10 Muxes, (24T,12S,94ios)	12712+4927=17639	6347
Promote(*mgcd*, {-})	1 FU (alu), 4 Regs, 6 Muxes, (7T,2S,148ios)	5093+719=5815	597 (467)
Extract(Promote (*mgcd*,{-}),{-})	1 FU (alu), 4 Regs, 5 Muxes, (6T,2S,145ios)	4744+604=5348	-

Notes: Column structural features: lists the main contents of the structure: number and names of FUs (between parenthesis), number of registers, number of multiplexers and features of the controller (number os transitions(T), number os states(S) and the sum of input and output ports (ios) of the controller). Column area: estimations of area in NTBR's - number of transistors - for the data-path, the controller and the total. Column gain(loss): shows potential gains or losses in using the transformed FUs, in NTBRs.

Additional experiments were carried out to observe the changes on hierarchical descriptions when transformations are applied on them. In table 3, the comparison between two hierarchical benchmarks, *pcl* and *pid*, and their transformed versions is shown; only the synthesised structural data are presented. Both transformed versions were obtained from the original ones by adding one extra transformed FU (*mmulmrep*+-=**Promote**(**Merge**(*mmul*, *mrep*),{+-}) into theirs set of architectures. Results in table 3 show that, in all cases, area reductions were obtained; the total area of FUs were always smaller. However, the controller size increased due to the increased complexity in accessing promoted operations, which uses more cycles as commented in section 4; AMICAL created controllers with more transitions due to this fact, it also implies that the new architectures are slower.

Table 3. Structural characteristics for the hierarchical benchmarks

	structural features	**area**	**reduction**
pcl	26Regs, 8FUs {ram,bset,mmul, mrep,alu, bmask(2),shift}, 16Muxes, (110T,48S,196ios)	52242+15254= 67496	
pcl*	26Regs, 6FUs {ram, bset, mmulmrep+-, bmask(2), shiftr}, 15Muxes, (120T,48S,192ios)	44925+16309= 61234	9,3%
pid	14Regs, 4FUs {rom, mrep, mmul, alu}, 8Muxes, (52T,22S,45ios)	40378+1758= 42136	
pid*	14Regs, 3FUs {rom, alu, mmulmrep+-}, 7Muxes, (56T,22S,42ios)	34125+1778= 35903	14,8%

Note: The benchmarks marked with a star are the transformed ones. See also notes in table 2.

6. CONCLUSION

We have presented a set of Architectural Transformations and a suitable modelling for hierarchical algorithmic descriptions. The main application for our ATs is for obtaining optimised hierarchical RTL structures from hierarchical algorithmic descriptions, via HLS. By using such optimising tools, designers are freed to write hierarchical algorithmic descriptions without worrying over some implementation issues like consequences of writing style on structural level quality. We have also presented some experimental results which confirm such a claim.

REFERENCES

[1] D.D.Gajski, N.D.Dutt, A.C.H.Wu e S.Y.L.Lin, "High-Level Synthesis - Introduction to Chip and System Design", Kluwer Academic Publishers, 1992.

[2] W. Geurts, F. Catthoor, H. De Man: Time Constrained Allocation and Assignment Techniques for High Throughput Signal Processing. Proceedings of the 29th Design Automation Conference, pp.124-127,1992.

[3] D. Sreenivasa Rao, F. J. Kurdahi: Controller and Datapath Trade-offs in Hierarchical RT-Level Synthesis. Proceedings of the 7th International Symposium on High-Level Synthesis, 1994.

[4] R. Genevriere, R. Camposano: Partitioning and Restructuring Designs on the Behavioral Level. Technical Report SFB358-B2-11/94, University of Paderborn, Technical University of Dresden, 1994.

[5] P. Kission, H. Ding, A.A.Jerraya: Structured Design Methodology for High-Level Design. Proceedings of the 31th Design Automation Conference, pp.466-471, 1994.

[6] O. Bringmann, W. Rosenstiel: Cross-Level Hierarchical High-Level Synthesis. Proceedings of the 1998 Design Automation and Test in Europe.

[7] J.C.Wang, M.Y.Teruya, J.V.Vale Neto, M. Strum, A.A.Jerraya: A Recursive High Level System. Proceedings of the 5th International Conference on VLSI and CAD - Seoul, South Korea, pp. 412-414, 1997.

[8] B.G.Hald, J.Madsen, A.A.Jerraya: A New Approach to Optimization and Reuse of Hierarchical Architectures. Accepted for publication by IEEE Transactions on VLSI.

[9] A.A.Jerraya, I.Park, K.O'Brien, "AMICAL: An Interactive High-Level Synthesis Environment", Proceedings of European Design Automation Conference, Paris, France, 1993.