HIDE+: A Logic Based Hardware Development Environment

AbdSamad BenKrid[*], *IEEE Member*, Khaled Benkrid[**], *Senior IEEE Member*

[*]School of computer science, The Queen's University of Belfast, Malone Road, BT 7 1NN, Northern Ireland, UK

[**]School of Engineering and Electronics, The University of Edinburgh, Mayfield Road, EH9 3JL, Scotland, UK

Abstract— *With the advent of System-On-Chip (SOC) technology, there is a pressing need to enhance the quality of design tools available and increase the level of abstraction at which hardware is designed, implemented and programmed. This would reduce the gap between what is currently achievable technologically, and what hardware engineers are capable to produce given time to market constraints. Hardware development should hence become easier and less time consuming, without scarifying the implementation efficiency. Towards this goal, we present in this paper a simple structural high-level hardware language called HIDE+, particularly suitable for the rapid generation of highly parameterised, and highly efficient, hardware cores. We detail the syntax and semantics of HIDE+ and illustrate how highly scaleable, parameterised and optimised architectures can be described and automatically generated from it, using a small set of constructors. HIDE+ offers a much more abstract way of describing hardware than is possible with traditional hardware description languages such as VHDL or Verilog. Although less abstract and extensive than other electronic system language environments, HIDE+ does not compromise on hardware efficiency. It can thus be of great use to SOC design as an Intellectual Property (IP) development environment.*

I. Introduction

Recently, semi-conductor technology has made spectacular advances leading to high-density fabrication and the incorporation of hybrid technologies on a single chip [1]. Nevertheless, the design productivity of engineers has not kept pace with these advances [2]. To close this gap and meet the stringent time-to-market and other constraints, there is a pressing need for higher quality hardware design tools and associated methodologies and design flows. To this end, researchers have recently introduced various System Level Design Languages (SLDLs) to raise the design abstraction level e.g. by focusing on system behaviour rather than low-level implementation details [3][4][5]. However, researchers recognise that the key to cope with the complexities involved with System-On-chip (SOC) design remains the reuse of Intellectual Property (IP) cores. Nonetheless, the use of third-party IP is fraught with problems to cost of ownership, lack of proper documentation, and maintenance issues. Moreover, IP cores integration, especially if these come from different providers, is not straightforward and increase the overall design time remarkably due to lack of standards [6]. Hence, the development of hardware design environment for IP cores generation, perhaps for in-house use, is one way to address the above problems.

Towards this end, we describe in this paper a high level Prolog-based hardware design environment, called HIDE+, specifically for the design of highly optimised DSP hardware architectures. The programming environment is built upon the success of a hardware description environment, developed by the authors, called HIDE [7]. Since its first publication, two major versions of HIDE have been developed to date. The details of these can be found in [8-9].

The remainder of the paper is organised as follows. Section 2 gives a brief overview of the HIDE environment on which HIDE+ builds. The rationale behind the development of HIDE+ is then given in section 3. Section 4 details the bases of the HIDE+ language, its structure and development environment, and illustrates this in the context of a number of DSP architectures. Conclusions will be drawn at the end.

II. Overview of the HIDE Environment

Figure 1 presents a block outline of the HIDE environment [9]. The environment has two libraries: the Basic Component Library (BCL) and the Object Description Library (ODL). The BCL contains netlist code for a large set of basic building blocks e.g. 1-2 bit adders, used to build more complex architectures. These are pre-designed by the *architecture builder*. The ODL on the other hand contains a header description of each of the BCL elements. The header description has the following format:

is_basic_block (*name, control_list, ports_list,…*)

where *name* denotes the name of the block and *control_list/ports_list* the list of its control/data ports. The headers are used by the HIDE engine when assembling their associated components. The *application developer* describes his/her architecture using the HIDE constructors given in [9]. The architecture description is then translated by the HIDE parser into a hardware configuration, which is then

(Advance online publication: 20 August 2008)

synthesised into EDIF or VHDL format. The details of the translation can be found in [9].

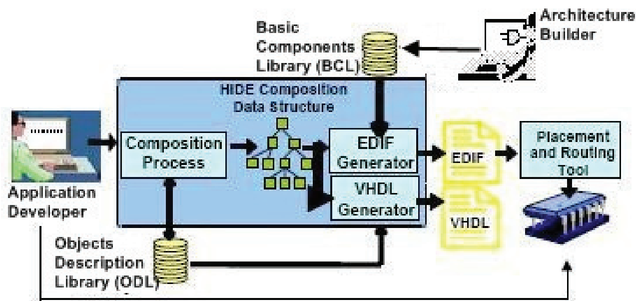

Fig. 1 Overall HIDE environment

HIDE+ inherits from HIDE its programming environment. However, the HIDE+ syntax is different from HIDE's in order to overcome some of the latter's limitations, as will be explained in the following sections.

III. HIDE+ Rationale

As stated in [9], the current version of HIDE is more suited to the description and synthesis of *regular* hardware architectures. However, many parallel hardware architectures are irregular. In that case, the HIDE simple block placement based on *horizontal* and *vertical* constructors is not appropriate. Furthermore, HIDE's control mechanism is basic and does not allow for the description of complex clocking schemes for instance. It also implicitly assumes integer data processing i.e. fractional numbers with rounding and truncation are not supported. Finally, HIDE does not separate between block connectivity and placement, as placement is implicitly assumed in HIDE's constructors e.g. *horizontal*, *vertical*.

The above limitations led us to develop a new notation, which builds on the advantages of HIDE, which are:

- The use of Prolog as the base development language, which allows for easy and smart coding of design rules for instance
- The use of a VHDL/EDIF generator engine which takes a high level abstract syntax tree as input
- Use of hardware skeletons as a hierarchical way to develop efficient hardware at increasingly abstract levels, while eliminating the aforementioned limitations of HIDE.

The following section describes the bases of HIDE+.

IV. Bases of HIDE+

To implement an architecture using HIDE+, the following three aspects need to be considered:

- **Architecture Description** to instantiate the architecture components, and specify their composition and interconnection
- **Architecture Control** to feed the architecture components with the appropriate control signals (e.g. clocking)
- **Architecture Constraints** to apply placement, routing, and timing constraints on the generated hardware configuration.

This approach allows a highly modular development environment. The following explains each of the above aspects in turn.

A. HIDE+ Architecture Description

The basic component in a HIDE+ configuration description is a *block*. The latter can be either a *basic* block or a *compound* block. However, unlike in HIDE, HIDE+'s does not group the block's ports into four sides regardless of their types [8]. Instead, HIDE+ considers a basic block as an *operation* or a *control* node to be included in a *data and control flow graph* representation of a compound block or architecture. The block is fed with a set of input data and control signals and generates output data through output ports, which would then be input data or control signals for other nodes in the graph.

Fig. 2 depicts a diagram of the *block* abstraction in HIDE+. The *block* ports are divided into two groups:

- Inputs: data inputs and carry-in input
- Outputs: data outputs and carry-out output

The blocks' control ports are treated separately by the architecture control engine (see section C below).

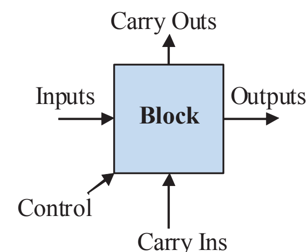

Fig. 2 HIDE+ block abstraction

All of the properties of a port are grouped into one constructor:

port*(Type, Name)***,** for a single port

or

port*(Type, Name(Dim))*, for a bus

where *Name* denotes the name of the port and *Type* its type (e.g. *in* for input, *out* for output). The number of the port wires can be any non-zero positive integer and is represented by *Dim*.

A.1. Block Interconnection

Fig. 3 gives the HIDE+ constructors used to build a *compound* block from elementary sub-blocks. When connecting two blocks ($Block_1$ and $Block_2$), the flow of data can be:

- from the outputs of $Block_1$ to the inputs of $Block_2$. The **serie (**or **s_seq** when replicating the same *block***)** constructor is used to annotate this connection
- from the outputs of $Block_2$ to the inputs of $Block_1$, or even from outputs of $Block_1$ back to its inputs. This feedback connection is specified using the **loop** constructor
- from the carry-out of $Block_1$ to the carry-in of $Block_2$. The two *blocks* are aligned in parallel without connecting their

data ports. The **parallel_with_carry** (or **p_seq_with_carry** when replicating the same *block*) constructor is used to annotate this connection

- from the carry-out of $Block_1$ to the carry-in of $Block_2$ along with connecting the data ports of the two *blocks*. The **serie_with_carry** (or **s_seq_with_carry** when replicating the same *block*) constructor is used to annotate this connection

On the other hand, the constructor **parallel** (or **p_seq** when replicating the same *block***)** is used to align the data ports of sub-blocks in parallel (so they can be grouped under one entity) without connecting their data ports and carry logic inputs outputs.

As in HIDE [8], the network connection constructor (**nc**([])) is used in conjunction with the above constructors if the user wants to connect the *blocks'* ports explicitly rather than relying on the *automatic* ports matching [8]. However, in HIDE+, the **nc**([]) constructor also handles buses instead of just single wire ports.

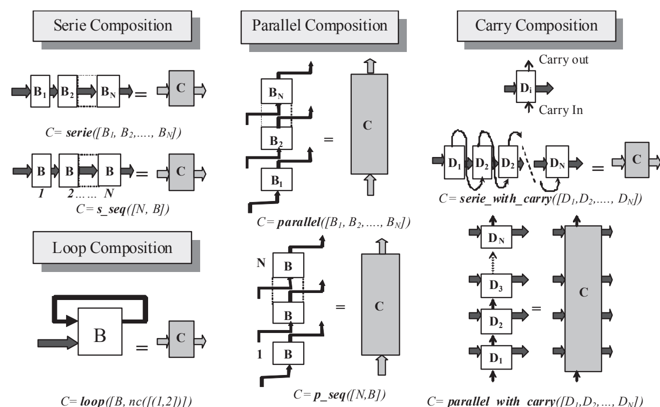

Fig. 3 HIDE+ basic constructors

Note that the above constructors have been derived after a rigorous exploration of a high number of DSP architectures, with the aim of allowing hardware designers to describe their architectures precisely and concisely.

A.2. Types of Blocks

In HIDE+, a hardware *block* can be:

- **Purely Combinatorial:** where the *block* is not clocked
- **Non-combinatorial:** where the *block* is clocked. Three cases are then possible:
  - The *block* is clocked with the architecture's master clock
  - The *block* is clocked with a clock rate *N* times lower than the master clock rate. Then, if this clock frequency is already available on the chip, it can feed directly the *block*'s clock input. Otherwise, the *block* needs to be enabled every *N* cycles of the Master clock. In the latter case, the clock *Enable* signal is generated automatically by the HIDE+ using a simple *counter* of period N
  - The *block* is clocked with a clock rate *N* times higher than the master clock rate. Then, the *block's* clock input should be driven by HIDE+ from dedicated chip logic (e.g. DCM block in Xilinx FPGAs [10]).

To portray the above three cases, a *ClkType* attribute is attached to every *block* constructor. It is equal to:

- "~": when the *block* is combinatorial
- "N": when the *block* is clocked with a clock rate of N-scale the master clock rate.

A.3. Rounding off

As shown in [11], to limit the unnecessary growth of hardware architectures' internal wordlength, *rounding off* and/or *truncation* are often carried out. The operation of *rounding off* a binary number B ($=b_{n-1}b_{n-2}\ldots b_0b_{-1}b_{-2}b_{-3}..b_{-(m-1)}$) at the order I consists of adding the bit $b_{-(I+1)}$ to the binary number B_T ($=b_{n-1}b_{n-2}\ldots b_0,\ b_{-1}b_{-2}b_{-3\ldots}b_{-I}$). Unlike the truncation operation, the implementation of the rounding operation normally requires dedicated logic: an adder (rounder) to add the carry bit $b_{-(I+1)}$ to the number B_T. However, this dedicated logic might not be needed if the operand to be rounded happens to be an input to an adder/subtractor in the architecture as the adder/subtractor own logic can be used to implement the rounding off operation of one operand (by feeding its $b_{-(I+1)}$ bit to the carry-in input of the adder), hence precluding the need for dedicated rounder hardware. Nonetheless, if both operands of an adder/subtractor need to be rounded off, the rounding bit of the second operand has to be delayed to feed the carry in of a next available adder/subtractor in the architecture, otherwise a dedicated rounder needs be inferred (see Fig 4, for an example of adder tree). HIDE+ automatically generates the necessary rounders and the rounding delays networks as well as feeding the carry in with the operands' rounding bits. To support this operation, the attribute *round(I)* (where *I* denotes the precision) is added to the relevant HIDE+'s arithmetic block header description.

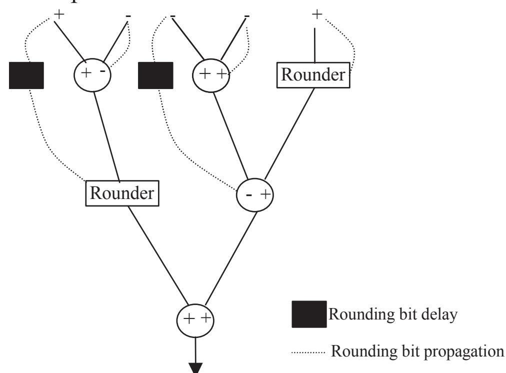

Fig. 4 Rounding-off scheme illustration in a 5-operand adder tree

It is worth noting that the rounding precisions of the operands don't have to be equal. As such, a *block Left* and *Right* rounding attributes need to be added. However, throughout this paper, we limit the presentation to a uniform rounding for the sake of simplicity.

B. HIDE+ Library Structure

The HIDE+ library contains a range of components with different levels of abstractions (e.g. from a 1-bit section of a

multiplier to a fully parameterised multiplier unit). The library is built hierarchically as illustrated in Fig. 5.

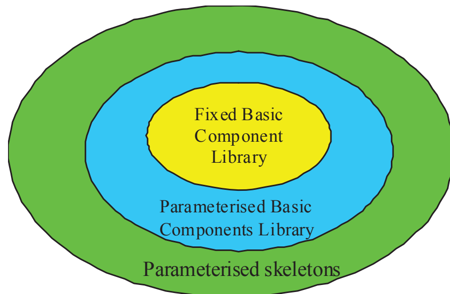

Fig. 5 A layered model of the HIDE+ library

The following sub-sections detail each layer in turn.

B.1. Fixed Basic Components Library

At the bottom level lays the fixed Basic Components Library (BCL). This groups the basic building *blocks* that implement the widely used operations in DSP applications, such addition, multiplication, delay, etc but at one bit level. These *blocks* are described in VHDL and EDIF format 2.0. This layer implemented by the architecture builder constitutes the nucleus of any HIDE+ constructed *block*. The properties of this layer's *blocks* are stored in the Object Description Library (ODL) (see section II) so that the user can instantiate them when composing DSP operators.

Most of this layer's components have three varieties: combinatorial, clocked with the master clock or a dedicated clock, or instead at each N cycles of the master clock so the clock enable is used.

B.2. Parameterised Basic Components Library

This layer delivers the basic DSP operations (e.g. N bits buffers, adders, etc). The blocks are fully parameterised, and composed solely from the fixed BCL components. The following gives examples from this layer's components.

- **Line Buffer**

Buffer units are often used in order to synchronise the supply of data. A buffer unit of size *Size* words and I/O wordlength *WL* is obtained by invoking the following constructor:

lb*(Size, WL, ClkType)*

- **Multi-Lines Buffer**

A Multi-Lines buffer is used to generate parallel data outputs from a serial stream of samples as shown in Fig. 6. This configuration can be generated by invoking the following constructor:

ser2Par*(NumOfPorts, WL, Size, ClkType, Flag)*

where *NumOfPorts* specifies the number of lines to be buffered each of *Size* words length. *Flag* is a Boolean variable that specifies whether the input sample should also be forked to the output or not.

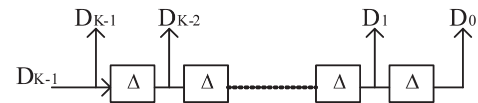

Fig. 6 A serial to parallel converter: ser2Par(K)

- **Truncation**

Truncating an input of *InWl* bits to *Prec*-bits precision is obtained by invoking the following constructor:

truncator*(InWl, Prec)*

- **Adder/Subtractor**

This performs a *weighted* addition/subtraction since the operands of an addition/subtraction might need to be shifted before addition/subtraction [8] (see Fig. 7). The required constructor for the adder is:

adder*(OutWl, LeftOff, RightOff, ClkType, Round)*

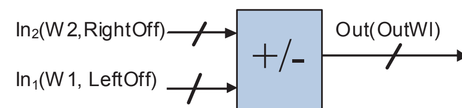

Fig. 7 A weighted adder/subtractor

- **Counter**

This block is useful in generating a regular periodic sequence. It is created by calling the following constructor:

counter(*UpOrDown, Step, InitSate, Period, ClkType, TypeOfOut*)

where:
UpOrDown: specifies if it is an upward or downward counter
Step: specifies the step-size of the counting
InitState: specifies the initial value of the counter
Period: specifies the period of the counter
TypeOfOut: a Boolean flag that specifies if the combinatorial output of the counter should also be available at the output.

- **LUT**

This is useful to configure FPGAs' Lookup-Tables. For instance, a 6-input LUT can be invoked by calling the constructor:

lut6*(INIT)*

where INIT specifies the function of the LUT.

B.3. Parameterised Skeletons Library

This layer contains higher level units called *skeletons*. A skeleton is a *mapped-to-structure*, to which the user can supply not only fixed and parameterised BCL components but even other skeletons, as parameters. Skeletons embed *optimisation* rules for logic use reduction and speed enhancement. An example of such rules computes the minimum wordlength needed at every node of an architecture [12]. This minimum wordlength depends on the dynamic range of the node's operands as well as the node operation itself. For instance, the following constructor is inherently

called inside a coefficient-multiplier skeleton code to determine the minimum wordlength necessary at its output:

nodeDynamic*(Coeff, Scale, InValue, OutValue, OutWl)*

where *Coeff* denotes the multiplier operand's coefficient value, *Scale* any scaling applied to the multiplicand, *InValue* is a 2-tuple (Min, Max) representing the dynamic range of the input, *OutValue* is a 2-tuple representing the dynamic range of the output and *OutWl* is the minimum wordlength necessary to cover the output's dynamic range. Similarly, the following constructor is invoked to find the minimum wordlength at the output of a generic node:

nodeDynamic*(Op, Scale, InValue, OutValue, OutWl)*

where *Op* can be either *add* (for an addition), *sub* (for a subtraction), *max* (for a maximum) or *min* (for a minimum), *Scale* is a tuple representing any weighting applied on the operands and *InValue* is a list of tuples representing the dynamic ranges of the operands. The *outValue* represents the *InValue* attribute of the subsequent node in the architecture. Hence the above constructors can be applied iteratively through all of the architecture's nodes.

The following gives illustrative examples of skeletons:

- *Reduction Tree skeleton*

A reduction tree skeleton reduces a set of operands into one result. Fig. 8 gives an example of 2-input node tree structure.

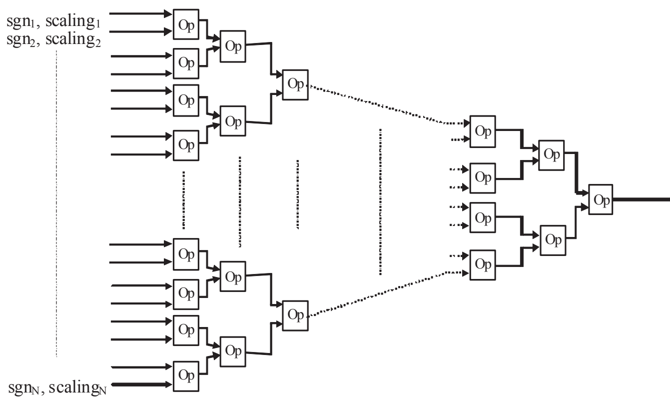

Fig. 8 A reduction tree skeleton

To generate an instance of this skeleton, the following constructor is provided:

tree*(Op, NodeSz,TreeSz, Round, ClkType, OrdOfPip, InpVal, BoundWL, OutVal, OutWl, SgnOfOp, Scaling)*

where *Op* specifies the tree operation (i.e. add, max or min), *NodeSz/TreeSz* the node/tree number of inputs, *OrdOfPip* the tree pipelining depth, *BoundWL* the upper bound wordlength set by the user if any, and *SgnOfOp/Scaling* the sign/scaling of operands.

This constructor embeds rules to generate the final hardware configuration with the required wordlength at every node of the architecture as well as the rounding, clocking and pipelining.

- *Reduction Chain Skeleton*

A reduction chain skeleton reduces a set of operands into one result. Fig. 9 gives an example of 2-input node chain structure.

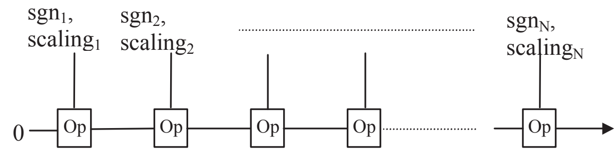

Fig. 9 A 2-input node reduction chain skeleton

To generate an instance of this skeleton, the following constructor is provided:

chain*(Op, NodeSz,TreeSz, Round, ClkType, OrdOfPip, InpVal, BoundWL, OutVal, OutWl, SgnOfOp, Scaling)*

- *1-D FIR Filter Structures*

The following explains how general and symmetric FIR filters are described in HIDE+.

a) General FIR filter

The architectures of a general FIR filter have mainly two forms: direct and inverse [13].

1. Direct Form Filter Structure

Fig. 10 shows the direct form of a 1-D FIR filter.

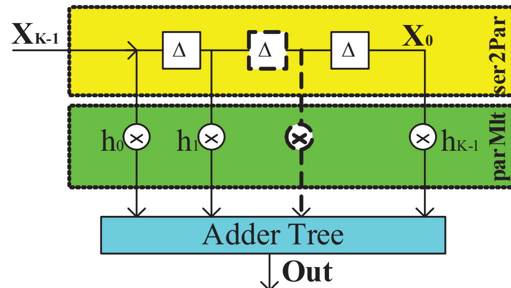

Fig. 10 Direct form structure of a K-taps FIR filter

Its HIDE+ description is as follows:

```
Op = add
B1 = ser2Par(K-1, InWl, 1,ClkType, true),
// parMlt is implemented using parallel constructor
B2=parMlt(FltCoefsList,Coefs_Wl,ClkType,Mlts_OrdPip,InVal,
B2_OutVal, B2_Wl),
B3 = p_seq(K, truncator(B2_OutWl, Prec)),
B4 = tree(Op, NodeSz, K, Round,ClkType,OrdOfPip, B2_Out_Val,
BoundWl, OutVal,OutWl, SgnOfOp, Scaling),
Constructor=serie([B1,B2,B3,B4]),!.
```

where *FltCoefList/Coefs_Wl* gives the filter's coefficient values/wordlength, *Mlts_OrdPip* specifies the pipelining order of each multiplier, and *B2_OutVal* is a 2-tuple list of the dynamic ranges at the multipliers' outputs.

2. Inverse Form Filter Structure
Fig. 11 shows the inverse form structure of the 1-D FIR filter.

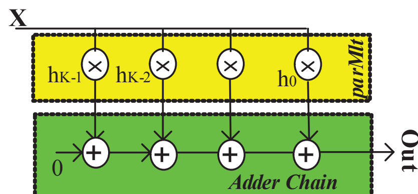

Fig. 11 Inverse form structure of a K-taps FIR filter

Its HIDE+ description is as follows:

```
Op = add
B1 = nc([(1,1),(1,2),…,(1,K)]),
B2=parMlt(FltCoefsList, Coefs_Wl, ClkType, Mlt_OrdPip, InVal,
B2_OutVal, B2_Wl),
B3 = p_seq(K, truncator(B2_Wl, Prec)),
B4 = chain(Op, NodeSz, K, Round, ClkType, OrdOfPip, B2_OutVal,
BoundWl, OutVal, OutWL, SgnOfOp, Scaling),
Constructor = serie([B1,B2,B3,B4]),!.
```

b) Symmetric FIR Filter
The architectures of a symmetric FIR filter have mainly two forms: direct and inverse.

1. Direct Form Symmetric Structure
Fig. 12 shows the direct form of a 1-D symmetric FIR filter.

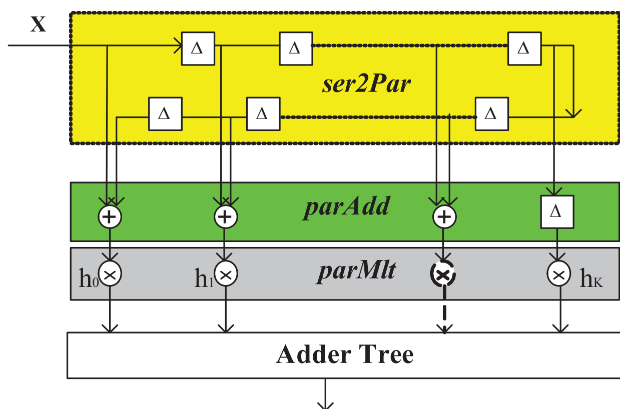

Fig. 12 Direct form structure of a (2K+1)-taps FIR filter

The HIDE+ description of a 1-D L-tap direct symmetric filter is:

```
Op = add
B1 = ser2Par(L-1, InWl, 1, ClkType, true),
B2 = generate_nc_for_symmetric filter,
// parAdd is implemented using parallel constructor
B3=parAdder(InVal, ClkType, B3_OutVal),
 ( (even(L), Half is L/2); (Half is L//2+1) ),!,
B4 = p_seq(Half, truncator(B3_Wl,Prec))
B5=parMlt(FltCoefsList, Coef_Wl, ClkType, Mlts_OrdPip,
B4_OutVal, B5_OutVal, B5_Wl),
B6 = tree(Op, NodeSz, Half, Round, ClkType, OrdOfPip,
B5_OutVal, BoundWL, OutVal, OutWl, SgnOfOp, Scaling),
Constructor = serie([B1,B2,B3,B4,B5,B6]),!.
```

2. Inverse Form Symmetric Structure
Fig. 13 shows the inverse symmetric form of the 1-D FIR r.

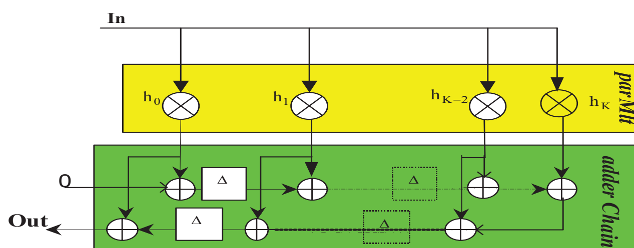

Fig. 13 Inverse form structure of a (2K+1)-taps symmetric FIR filter

The HIDE+ description of a 1-D L-tap inverse symmetric filter is:

```
Op = add
B1 = nc([(1,1),(1,2),…,(1,L)]),
B2 = parMlt(FltCoefsList, Coef_Wl, ClkType, Mlts_OrdPip, InVal,
B2_OutVal, B2_Wl),
B3 = p_seq(L, truncator(B2_Wl, Prec))
B4 = chain (Op, NodeSz, L, Round, ClkType, OrdOfPip,
B2_OutVal, BoundWl, OutVal, OutWL, SgnOfOp, Scaling),
Constructor = serie([B1,B2,B3, B4]),!.
```

- **2-D FIR Filter structure**

Fig. 14 shows the architecture of a (*K*x*M*) taps 2-D FIR filter. The 2-D FIR filter is implemented by means of *K* 1-D FIRs (each one having *M* taps) and "*K*-1" row delays. Each row delay holds a whole image raw (e.g., N pixels for an NxN image).

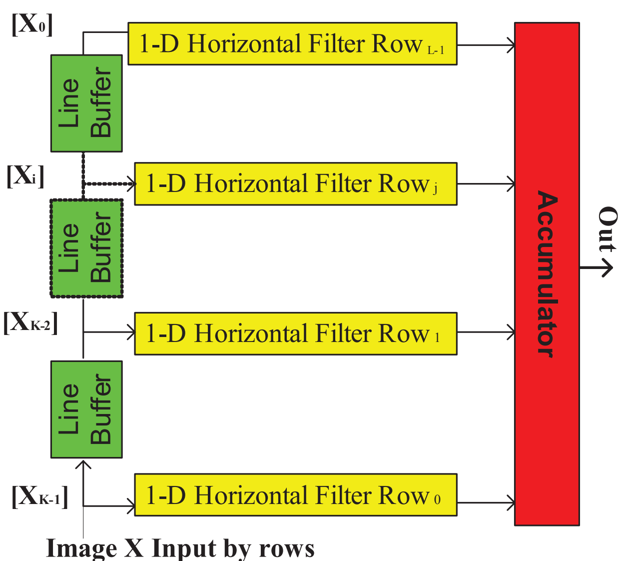

Fig. 14 (*KxM*)-tap 2-D FIR filter architecture

The following presents the main HIDE+ constructor calls to implement a 2-D KxM FIR filter:

```
Op = add
B1 = ser2Par(K, InWl, RowSz,ClkType, true),
// parFlt is implemented using parallel constructor
B2 = parFlt(RowFlt, FirStructures, B2_OutVal, B2_OutWl),
B3 = tree(Op, NodeSz, M, Round,ClkType,OrdOfPip, B2_OutVal,
BoundWl, OutWl, ones(1,M),ones(1,M)),
Constructor=serie([B1,B2,B3]),!.
```

where *FirStructures* instantiates one of the 1-D FIR structures shown in the previous section.

Note that this skeleton architecture has been used to implement a generic image algebra neighbourhood operation core set [8], simply by replacing the multiplier by the required local operator (e.g. addition) and the adder by the global operator (e.g. maximum, minimum). The implementation configuration delivers the same performance as when optimised carefully by hand [14].

C. HIDE+'s *Architecture Control*

In addition to the provision of regular counter constructor (see section B.2), HIDE+ is able to generate the logic for any *periodic pattern* output. This logic consists of LUTs associated with flip flops where the flip-flops' outputs are fed back into the LUT inputs. Fig. 15 shows, for example, the required logic for implementing the periodic sequence [0,1,0,0].

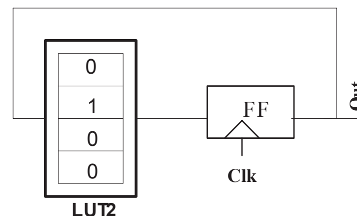

Fig. 15 Implementation of a [0,1,0,0] periodic sequence generators.

The equivalent HIDE+ constructor is:

B=**loop**([**serie**([lut2(4),FF])])

More complex controllers are implemented via Finite State Machine (FSM) structures. These are automatically generated using high level HIDE+ constructors as explained below.

Fig. 16 shows the general structure of an FSM [15]. The *current state(value)* of the machine is stored in the *state memory* (a set of *n* flip-flops or a memory). The machine's *next state* is a function of the *current state* and the *inputs*. The outputs in *Mealy* FSMs are a function of the *current state* and the *inputs* while in the *Moore* FSMs, outputs are a function of the *current state* only.

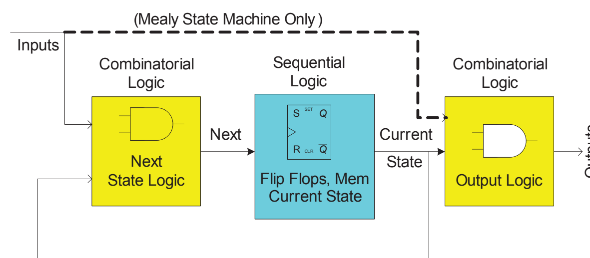

Fig .16 FSM Block Diagram

Usually, FSM are described by state diagrams. State diagrams are then converted into *state* and *output* tables from which the structure of the *next state* and *output* circuits can be derived. The current version of HIDE+ does not allow the designer to draw the state diagram graphically. Instead, he/she can set the *next state* function by invoking the following constructor:

stateTable[(*In,CurrentState,NextState*),….]

The designer can set the *output* function by calling:

outTable[(*In,CurrentState,outVal*)], in Mealy FSMs

or

outTable[(*CurrentState,outVal*)], in Moore FSMs

where *In, CurrentState, NextState,* and *outVal* can be given in decimal or binary representation.

Subsequently, the FSM block's architecture is generated by invoking the following constructor:

genFSM(*Type,StateTable,OutTable,EncType*)

where *Type* indicates the type of the FSM and *EncType* specifies the FSM' states encoding: Binary, Gray Code, or oneHot. The *genFSM* function implements the *Quine McLusky* algorithm needed to minimise the combinatorial logic [15]. Finally, a *drive_controls* constructor [7], connects the output of the above FSM *block* to the relevant block(s) in the sought architecture.

D. HIDE+ Architecture Constraint

In addition to mapping constraints applied on the BCL components, designers can attach *placement* and *timing* constraints to the designed hardware architecture. Currently, these are automatically passed to back-end synthesis tools. In the future, HIDE+ will include rules for automatic time-driven floorplanning.

V. Implementation Results

HIDE+ has been used to implement a wide range of real word applications on actual FPGA Hardware [8][9][14]. The results show that HIDE+ can deliver the same performance as the best handcrafted designs, with the added parameterisability and scalability features. This section gives a sample of these results through a Daubechies-8 FIR filter [16] implementation on Xilinx XCVE50-8 FPGA [17]. The filter is an instance of a HIDE+ core which was written with various optimisations embedded into it, including automatic minimum word length and precision inference, and efficient FPGA hardware mapping. Table 1 below gives the performance achieved by implementing the low Daubechies-8 FIR filter using the structures of figure 10 and 11, using both HIDE+ and a handcrafted schematic-entry design of the same filter. The filter coefficients have been represented in 8 bits and 2 fractional precision were allocated
to the internal wordlength.

	Area (Slices)	Speed (MHz)
Fig. 10 architecture	147	~167
Fig. 11 architecture	113	~159

(a) SchematicEntry

	Area (Slices)	Speed (MHz)
Fig. 10 architecture	147	~167
Fig. 11 architecture	113	~159

(b) HIDE+

Table 1. Performance of a low Daubechies-8 FIR filter implementation on Xilinx XCVE50-8 FPGA using schematic entry and HIDE+ tool

As can be seen from the table, the HIDE+ core delivers the same performance as a handcrafted schematic-entry design.

VI. Conclusion

In this paper, we have described the bases of a Prolog-based structural hardware development environment, called HIDE+, which allows for very concise and abstract descriptions of structured hardware architectures, and translates them automatically into very efficient hardware implementations. Based on a hierarchical library of hardware building *blocks* and a small set of constructors, we have illustrated the use of HIDE+ in the construction of a number of FIR-based architectures. These designs proved optimal in the sense that the same optimisations undertaken by hand were achieved automatically through the use of HIDE+. The achieved concise descriptions show clearly the benefit of the modular structure of the language in facilitating the development of efficient and reusable designs and IP cores in general (see section B.3).

The development of in-house Intellectual Property cores has become vital in the EDA industry, and with current high density heterogeneous hardware platforms and stringent time-to-market constraints, HIDE+'s approach to hardware development can become very appealing. And although the proposed environment does not include behavioural modelling currently, and does not allow for concurrent hardware software co-design, it provides a fully programming environment for the development of highly parameterisable and optimised IP cores. Nonetheless, the extension of HIDE+ and its integration to higher level SLDLs are currently being considered.

VII. References

[1] International Technology Roadmap for Semiconductors (ITRS), 2005, available at http://public.itrs.net/

[2] David August, Kurt Keutzer, Sharad Malik, Richard Newton. Programmable ASICs to reduce costs. EE Times, November 2000.

[3] System C Home page: http://www.systemc.org.

[4] Spec-C Home page: http://www.cecs.uci.edu/~specc/

[5] Celoxica Limited, Handel C information sheets. Available at http://www.celoxica.com.

[6] Harriet Harvey-Horn. IP Assessment: Issues and Strategies. Silicon Integration Initiative, August 1999.

[7] K.Alotaibi, "A high level hardware description environment for FPGA-based image processing applications," PhD Thesis, Department of Computer Science, The Queen's University of Belfast, 1999.

[8] K. Benkrid, D. Crookes, "From application descriptions to hardware in seconds: a logic-based approach to bridging the gap", IEEE Transactions on Very Large Scale Integration (VLSI) Systems, TVLSI12, vol. 4, 2004, pp. 420–436.

[9] K. Benkrid, S.Belkacemi, A. Benkrid, "HIDE: A Hardware Intelligent Description Environment", In Elsevier's Journal of Microprocessors and Microsystems, Special Issue on FPGA-based Reconfigurable

Computing, 30, Vol. 6, pp. 283-300, September 2006.
[10] *Virtex-5 Family Overview Platforms,* Xilinx Inc., 2007. Available: http://direct.xilinx.com/bvdocs/publications/ds100.pdf
[11] A. Benkrid, K. Benkrid, D. Crookes, "A Novel Approach for Diminishing and Predicting the Error Dynamic Range in Finite Wordlength FIR Based Architectures", IEEE International Conference on Acoustics, Speech, and Signal Processing (ICASSP'03), vol. 2, pp. 581-584, April 6-10, 2003, Hong Kong
[12] K. Benkrid, K. Benkrid, D. Crookes, "The Optimal Wordlength Calculation for Forward and Inverse Discrete Wavelet Transform Architectures", SPIE Journal of Optical Engineering, OE, Vol. 43, Issue 2, pp. 455-463, February, 2004.
[13] Peter Pirsch, "Architectures for Digital Signal Processing," John Wiley & Sons, 1999.
[14] A. Benkrid, "Design and Implementation of 2-D Discrete Wavelet Transforms on FPGAs", PhD thesis PhD Thesis, Department of Computer Science, The Queen's University of Belfast, 2003.
[15] S. Golson, "State Machine Design Techniques for Verilog and VHDL" Synopsys Journal of High-Level Design, September 1994, pp. 1-48.
[16] M. Vetterli, M. Kovacevic, *Wavelets and Subband Coding.* Prentice Hall, New Jersey, USA, 1995.
[17] *Virtex-E Family Datasheet*, Xilinx Inc. , 2003. Available: http://www.xilinx.com/support/documentation/data_sheets/ds022-1.pdf