# Technology driven High-Level Synthesis

M.Joseph
Department of Computer Engineering
National Institute of Technology Karnataka
Mangalore - India
Email: mjoseph_mich@yahoo.com

Narasimha B.Bhat
Manipal Dot Net Pvt Ltd
37. Ananth Nagar
Manipal - India
Email: narasim@manipaldotnet.com

K.Chandra Sekaran
Department of Computer Engineering
National Institute of Technology Karnataka
Mangalore - India
Email: kchnitk@gmail.com

***Abstract*— Technology driven High-Level Synthesis make the present High-Level Synthesis knowledgeable of the target Field Programmable Gate Array. All the functions of High-Level Synthesis become aware of target technology since parsing. It makes right inference of hardware, by attaching target technology specific attributes to the parse tree. This right inference will guide to generate optimized hardware.**

**Keywords:** High-Level Synthesis, Target Technology, Attribute Grammars, Optimization, FPGA.

## I. Introduction

In Very Large Scale Integrated Circuit (VLSI) design, higher level abstraction of the circuit design is inevitable due to 1. Higher complexity, 2. Shrinking device size and 3. Shorter time to market. Hardware Description Language (HDL) allows, representation of a digital synchronous system at a higher abstraction level. High-Level Synthesis (HLS) converts this HDL input into corresponding Register Transfer Level (RTL) netlist. This netlist will further be technology mapped and implemented onto a Field Programmable Gate Array (FPGA) by down stream implementation tools. Optimization at this level is very much necessary to design a chip which consumes less silicon area and power and also works at higher speed. It is possible by applying compiler optimization techniques onto the intermediate representation and also by improving the design methodology itself. This paper suggests a new approach in the existing HLS design methodology for optimal hardware generation since compiler optimization techniques offer only little improvement [23].

Present generic HLS approach is independent of target technology, onto which the circuit is to be implemented. Due to this the potential of the target technology is not exploited and it leads to sub-optimal generation of hardware. We propose a new methodology to make the present HLS approach aware of the target technology using Attribute Grammars (AGs), so that it generates the optimized hardware, by exploiting the technology to its potential. Technology driven HLS (THLS), which uses this new methodology, is the tool developed by the authors.

### A. High-Level Synthesis

HLS takes a source program in any HDL as input and converts it into RTL structures. Its front end includes scanner, parser and intermediate code generator. Parser converts the syntax of the HDL input into an annotated parse tree that is then elaborated into an intermediate representation. Elaboration process instantiates modules, evaluates and propagates symbolic constants, checks the connectivity of all the devices and produces a checked consistent design. The Intermediate Representation (IR) is Control/Data Flow Graph (CDFG), a variant of syntax tree along with control information. Its back end consists of optimizer and hardware generator (synthesizer) phases which are scheduling and allocation. The optimizer applies compiler optimization techniques on the CDFG, to improve it, keeping speed, silicon area and power as optimization factors. Scheduling assigns operations to clock cycles. Allocation assigns operations to functional units like arithmetic logic units, multiplexers and storage elements [5], [6], [12]. It uses a generic library of devices like Library of Parameterized Modules (LPM)[25].

### B. Target Technology

There are two basic versions of Programmable Read Only Memories (PROM) available; one can be programmed by the manufacturer and the other by the end user. The former is Mask Programmable and latter is Field Programmable. The FPGA consists of programmable array of uncommitted elements, which can be interconnected in a generic way. Logic block is the basic unit of the FPGA that performs the combinational and sequential logic functions. Look Up Table (LUT), the logic block, is a digital memory with $k$ address lines that can implement any function of $k$ inputs by placing its truth table into the memory. The interconnect comprises segments of wire, where the segments may be of various lengths. Present in the interconnect are programmable switches that serve to connect the logic blocks to the wire segments or one wire segment to another. Logic circuits are implemented onto the FPGA by partitioning the logic into individual logic blocks as required via the switches. The structure and content of the interconnect in an FPGA is called its routing architecture consists of both wire segments and programmable switches [3], [26]. This FPGA is referred as *target technology* in this paper.

### C. Attribute Grammars

Attribute Grammars, a semantic formalism, devised by Knuth [1], attach attributes and semantic rules to the grammar symbols of a Context Free Grammar (CFG). Consider CFG $G = (S, N, T, P)$, where S is start symbol and $S \in N$; N is set of nonterminals; T is set of terminals and P is set of production rules. Let $X$, $X \in (N\ U\ T)$ can have a set of attributes $A$, and an attribute may be any context sensitive property of $X$, for instance, $value$ of $X$. The attributes can be either synthesized or inherited. Synthesized attributes of a grammar symbol, in a parse tree, depend upon the attributes of its children, whereas the inherited attributes depend upon the attributes of its parent and siblings of the corresponding parse tree. There can be a set of semantic rules for each production $pr \in N$, called $R$, which define the attributes in terms of other attributes of terminals and nonterminals of the same production [2].

The rest of the paper is organized as follows: Section 2 presents the related work done in making HLS aware of target technology. Section 3 gives details about the suggested methodology. Section 4 presents the implementation aspects. Section 5 discusses the result and implications and section 6 gives the concluding remarks.

## II. Related Work

Recently several approaches have been presented taking physical information into account. Most of the algorithms use floor planning information in HLS to estimate area and performance accurately [4], [8]. Stammermann et al proposed an approach, taking binding, allocation and floor planning information into account for low power in HLS [20]. Lot of techniques have already been proposed taking into account power consumption in HLS [10] - [17]. Some contributions also consider interconnect power [9], [14], [18]. Junhyung Um et al presented a new RTL synthesis approach for arithmetic circuits, which considers fast timing and easy layout. They proposed a two-phase approach, which (Phase 1) produces an optimal-timing arithmetic circuit and (Phase 2) refines the circuit structure obtained in Phase 1, can be used effectively in synthesizing data-paths [19]. Min Xuy et al addressed the problem of layout-driven synthesis as this step has a direct relevance on the final performance of the design [15]. Jason et al presented a complete bitwidth-aware HLS flow, including bitwidth analysis, simultaneous scheduling and binding [22]. Gwenole Corre et al presented a strategy to take into account the memory architecture and memory mapping in HLS for Real-Time VLSI circuits. They defined memory mapping constraint and included it in the scheduling algorithm [21]. Oliver et al presented an approach on combined HLS and partitioning for FPGA-based multi-chip emulation systems to synthesize a prototype with maximal performance under the given area and interconnection constraints of the target architecture. Interconnection resources were handled similar to functional resources, enabling the scheduling and sharing of inter-chip connections according to their delay [16]. HDL compiler in Xilinx Synthesis Tool (XST) [26], a leading commercial synthesis tool, parses the HDL code and extracts the **known generic functions** like multiplexers, memories and others. It then maps them onto the technology primitives by the technology mapper. In this approach target technology knowledge is used only at the mapper not at the compiler.

TABLE I
Difference between the THLS and HLS

| No. | THLS | HLS |
|---|---|---|
| 1 | Target specific output | Generic output |
| 2 | Retains designer's technology knowledge | Looses designer's technology knowledge |
| 3 | Uses TSIR | Uses generic IR |
| 4 | Uses TSL | Uses generic library |
| 5 | Reduces the gap between tool and target capabilities | Widens the gap between tool and target capabilities |
| 6 | Exploits technology | Doesn't exploit Technology |
| 7 | Output is optimal in terms of speed, power and silicon area | Output is sub-optimal in terms of speed, power and silicon area |

*(TSIR - Target Specific Intermediate Representation; IR - Intermediate Representation; TSL-Technology Specific Library)*

The above mentioned approaches in HLS considered only part of the physical information for some functions of HLS. These are independent of target technology at higher abstraction i.e. at parser level. We present a new methodology, which makes the HLS, aware of the target technology since parsing itself. All the functions of HLS are knowledgeable of target technology in this approach.

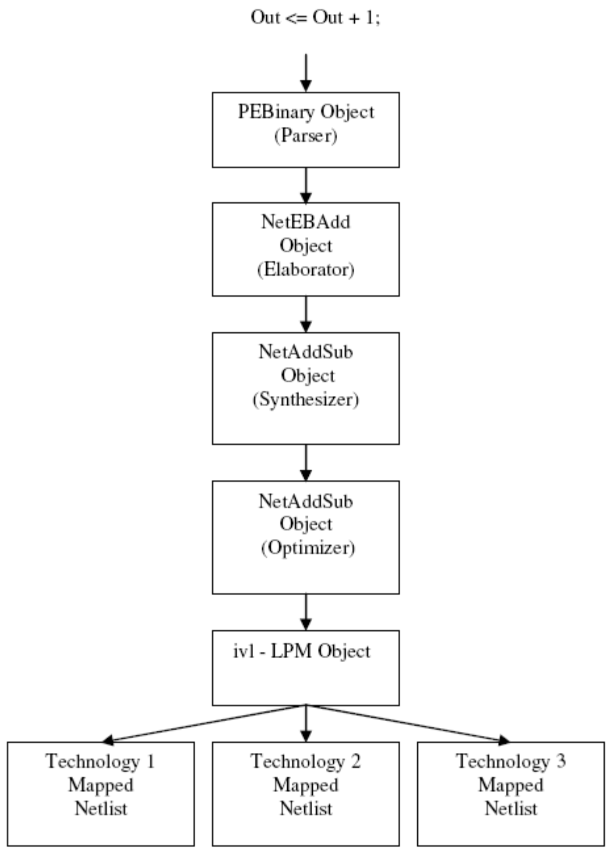
## III. Technology driven HLS

Present generic HLS approach checks only functional correctness of the design. It neglects the designer's technology knowledge coded into the design since it aims to generate generic output; e.g. it infers an adder for a count operation in the input. It looses the designer's intention of implementing a counter and also does not exploit the counter feature available in the target technology. This will result in sub-optimal implementation. If HLS is aware of target domain knowledge, optimal generation of hardware is possible [12]. To retain designer's technology knowledge coded into the design, and to exploit the target technology to its potential, HLS tool should be made aware of target technology.

THLS is a customized HLS tool for a particular target technology. All the phases of this tool are knowledgeable of the target technology. In THLS **attributes and target technology** are the key elements. Parser uses AGs to attach target specific attributes, to generate technology specific annotated parse tree. Elaboration then generates *Technology Specific Intermediate Representation* (TSIR) from this annotated parse tree. The optimizer then applies compiler optimization techniques on the CDFG, to improve it, keeping speed, silicon area and power as optimization factors. Synthesizer converts this TSIR into hardware. It uses the *Technology Specific Library* (TSL) not any generic library like LPM. Table 1. compares both the THLS and HLS approaches. Some salient features of this approach are discussed below.

### A. Right Inference in Parsing

THLS allows the parser to generate annotated parse tree, in which attributes are technology specific. *Attributes are*


Out <= Out + 1;
PEBinary Object (Parser)
NetEBAdd Object (Elaborator)
NetAddSub Object (Synthesizer)
NetAddSub Object (Optimizer)
ivl - LPM Object
Technology 1 Mapped Netlist
Technology 2 Mapped Netlist
Technology 3 Mapped Netlist

*ivl - Icarus Verilog*
*LPM - Library of Parameterized Modules*

Fig. 1. Compiler Transformations - Icarus tool

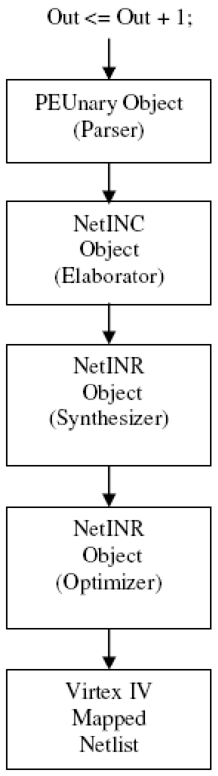

Out <= Out + 1;
PEUnary Object (Parser)
NetINC Object (Elaborator)
NetINR Object (Synthesizer)
NetINR Object (Optimizer)
Virtex IV Mapped Netlist

Fig. 2. Compiler Transformations - THLS tool

*abstractions of target technology features*. These attributes indirectly carry the cost information like speed, silicon area and power. Parser or elaborator use these attributes to infer the operations correctly. This guides to map right hardware devices during synthesis [24]. It also makes all the HLS functions to become aware of target domain.

### B. TSIR

TSIR is created based on the target technology. Technology details are embedded into the CDFG to make it technology specific. E.g: for shift register, there will be shift register node with the particular technology details.

### C. TSL

TSL is a library of technology specific devices, defined based on the target technology. E.g: for shift register, there will be shift register device as per target technology.

## IV. Compilation in THLS

We present the details of this new design methodology with few examples in the subsequent sections. IEEE standard 1364 HDL Verilog grammar is considered for the discussion.

### A. Upcounter

Consider the Verilog code segment '$out <= out + 1'$', which is a count operation. Parser in Icarus tool, converts it into a parser expression binary object $PEBinary$. Parser infers an addition operation here. Elaborator then converts this into a structural adder object $NetEBAdd$, a type of $NetEBinary$ object. $NetEBAdd$ is a node in the generic IR for addition operation. Synthesizer then transforms this into an adder device $NetAddSub$ from LPM. This ivl - LPM (Icarus Verilog LPM) is a generic object, which can be mapped to any target technology Technology 1 or Technology 2 and the like. Optimizer applies optimization techniques and improves this adder. Since this approach tries to synthesize for a generic architecture, it infers an adder for a counter, though counters are available in the target. Sequence of transformations for the code segment '$out <= out + 1'$', under Icarus tool is given in Fig.1.

Parser in THLS, on the other hand, converts the above code segment into a parser expression unary object $PEUnary$. Parser infers a count operation here. This makes elaboration to generate a structural counter object $NetINC$, a type of $NetEUnary$ object. $NetINC$ is a TSIR node. Synthesizer transforms this into a counter device $NetINR$ from TSL. Optimizer then improves it. Sequence of transformations for the code segment '$out <= out + 1'$ under THLS tool is given in Fig.2. Here target technology is considered as Virtex IV as an example case. *(Note: All objects referred in this paper are C++ objects.)*

## B. Shift Register

This example, shift register, is slightly different, where elaboration makes the inference. Consider the Verilog code segment '$reg \quad [15:0]sr'$, which is for shift register. Parser, in Icarus tool, calls the $make\_wire$ function with the attribute $identifier^1.name$, and create parser wire object $PWire$, for the identifier $sr$. It also creates a register variable entity. Parser uses another production rule for assigning the range for the register variable. It determines the range i.e. register width. Parser also creates a vector ($parser\_expr$) to store the Most Significant Bit (MSB) and Least Significant Bit (LSB) values of $sr$. Here 'MSB' and 'LSB' values are derived from their corresponding attributes. Elaboration associates *register variable* with $range$. Here, it infers a structural signal object ($sig$). Synthesizer then transforms this into a set of flip-flop devices from LPM. Optimizer applies optimization techniques on them. This is the present approach. This approach holds good for the Verilog source code segment '$reg \quad [7:0]out'$, which represents a register. For the above mentioned code segment '$reg \quad [15:0]sr'$, which represents a shift register, this approach will not hold good. This approach will implement it as a set of registers, not exploiting the shift register feature available in the target technology. This implementation is sub-optimal, since it occupies more silicon area and consumes more power.

Parser, in THLS, converts the same code segment into a parser wire object $PWire$. It also creates a register variable entity. Parser uses another production rule for assigning the range for the register variable. It determines the range i.e. register width. Parser also creates a vector ($parser\_expr$) to store the Most Significant Bit (MSB) and Least Significant Bit (LSB) values. Elaboration associates *register variable* with $range$. Then it infers a structural shift register object ($NetShiftreg$), a TSIR node. The choice between register and shift register is possible based on the width of the identifier. It is the difference between MSB and LSB values, which is dependent on target technology. Synthesizer transforms this into a shift register device from TSL. Optimizer then improves this.

## C. Memories

In target technology, distributed Random Access Memory (RAM), block RAM and Read Only Memory (ROM) features are available. At present, the HLS infers a generic memory at the compiler level, not making any choice, though it is clearly represented in the input. Consider the Verilog code segment '$reg\ mem[15:0]'$, which is an array to be implemented as memory in the target. In Icarus compiler, parser calls $make\_wire$ function, for the identifier, with the attribute $identifier^1.name$ and create a parser wire object $PWire$. It also creates a register variable entity. It sets the attributes '$expression^1.val'$ and '$expression^2.val'$, as index values, if they are constants. The attributes '$expression^1.val'$ and '$expression^2.val'$, which are array limits, represent its width. These index values will, later, guide to infer a generic memory $NetMemory$ during elaboration, which is a node in generic IR. Synthesizer will map it into a generic memory device $NetRamDq$ from LPM. This is the present approach. It will create complication and confusion in making a choice among memories for the down stream tools and lead to sub-optimal implementation sometimes. Consider the Verilog code segment '$reg\ mem[15:0]'$ in the source program, meant for a distributed memory of 16-bit in the target technology. The present tool has to rightly infer it as a distributed RAM, not a generic memory. In some other cases, a block RAM memory should be inferred. So it is imperative that the compiler should infer rightly and make a choice of memory at this abstraction itself.

In THLS, parser will infer the memory rightly at this abstraction itself. It is possible based on the width of the identifier given in the design, which is the difference between index values. This is the deciding factor to make choice between distributed RAM and block RAM; e.g. if the width is very large, it is better to infer a block RAM. The best choice for the code segment '$reg\ mem[15:0]'$ is distributed RAM of 16-bit. Here parser invokes the routine $make\_dis\_ram$ with corresponding attributes $identifier^1.name$, $expression^1.val$ and $expression^2.val$, based on the width of the identifier, which is dependent on target technology. Similarly it invokes the routine $make\_block\_ram$ with corresponding attributes $identifier^1.name$, $expression^1.val$ and $expression^2.val$, based on the width of the identifier, which is dependent on target technology. Later, elaboration will elaborate either $NetRamMemory$ or $NetBlockMemory$, which are TSIR nodes for distributed and block RAM entities respectively. Synthesizer will then map them into either $NetRamMemoryDevice$ or $NetBlockMemoryDevice$, which are TSL devices for distributed and block RAM entities respectively. *(Note: ROM is not considered here.)*

## V. Implementation Framework

We modified Icarus Verilog Compiler to develop the THLS tool. It is a compiler for the IEEE standard 1364 HDL Verilog. It translates the Verilog source code into RTL netlist formats for synthesis or other executable programs for simulation. The currently supported targets are $vvp$ for simulation and $xnf$ and $fpga$ for synthesis. It is an open source EDA (Electronic Design Automation) tool and also a part of gEDA (gnu EDA) [27].

We used Virtex IV, a high performance FPGA from Xilinx Inc, as the target technology. Its devices are user programmable gate arrays with various configurable elements. The innovative Advanced Silicon Modular Block column based architecture is unique in the programmable logic industry. Virtex-IV contains three families LX, FX and SX, which provide resources to develop logic, communication and digital signal processing applications. A wide array of hard IP-core

TABLE II
DEVICE UTILIZATION 1 FOR UPCOUNTER

| No. | Components | Icarus | THLS | ISE |
|---|---|---|---|---|
| 1 | LUT | 8 | 0 | 7 |
| 2 | XORCY | 7 | 8 | 7 |
| 3 | MUXCY | 6 | 7 | 7 |
| 4 | FDRE | 8 | 8 | 8 |

*(LUT - Look Up Table, XORCY - XOR gate with Carry, MUXCY - Multiplexer with Carry, FDRE - D-Flip-Flop)*

TABLE III
DEVICE UTILIZATION 2 FOR UPCOUNTER

| No. | Metric | Icarus | THLS | ISE |
|---|---|---|---|---|
| 1 | Gate Count | 101 | 95 | 115 |

*(Note: Gate count refers to Total equivalent gate count)*

TABLE IV
DEVICE UTILIZATION 1 FOR SHIFT REGISTER

| No. | Components | Icarus | THLS | ISE |
|---|---|---|---|---|
| 1 | SRLC16 | 0 | 1 | 1 |
| 2 | FDRE | 16 | 0 | 0 |

*(SRLC16 - 16-bit shift register with clock, FDRE - D-Flip-Flop)*

TABLE V
DEVICE UTILIZATION 2 FOR SHIFT REGISTER

| No. | Metric | Icarus | THLS | ISE |
|---|---|---|---|---|
| 1 | Gate Count | 128 | 72 | 72 |

*(Note: Gate count refers to Total equivalent gate count)*

blocks are also available. Its devices are produced on a state-of-art 90nm copper process using 300mm (12 inch) wafer technology. There is up to 40 speed over previous generation devices in the Configurable Logic Block (CLB) of Virtex-IV [26].

## VI. RESULT

We present here the result and analysis of two example Verilog programs, eight-bit upcounter and 16-bit shift register. We used Icarus, THLS and ISE synthesis tools for our experimentation. ISE is leading commercial synthesis tool from Xilinx Inc [26]. The product version used is ISE 8.2.01i. We executed the above mentioned example programs, in these three synthesis tools and generated synthesized netlists. We could generate programming files successfully in ISE 8.2.01i for the target Virtex IV. We use synthesis and map reports to compare and analyze the result of these three synthesis tools.

### A. Upcounter

Table 2. gives the device utilization details based on synthesis reports for upcounter implementation under these three tools. Icarus and ISE tools use the target technology components LUT, XOR gate with Carry (XORCY), Multiplexer with Carry (MUXCY) and D-Flip-Flop (FDRE) for implementing it. In Icarus, eight LUTs, which are meant for performing logic functions, are used for counter implementation. LUTs are powerful logic elements and they cannot be wasted like this. This implementation will occupy four slices in a CLB of Virtex IV. Similarly ISE uses 7 LUTs. On the other hand, THLS tool exploits the **carry-chain logic**, XORCY and MUXCY, meant for implementing counters in the target, leaving all the eight LUTs free. THLS tool offers three times improvement in silicon efficiency and four times improvement in speed over the Icarus tool.

Table 3. gives the details of total equivalent gate count based on map reports generated after technology mapping in ISE. It is a metric to evaluate the tool's silicon efficiency. THLS tool has 1.06 times improvement in silicon efficiency over Icarus tool and 1.21 times improvement over ISE.

### B. Shift Register

Table 4. gives the device utilization details based on synthesis reports for shift register implementation under these three tools. Icarus tool uses 16 FDREs, whereas the THLS tool uses single shift register (SRLC16) for the implementation. The former occupies eight slices of two CLBs in the target technology. The latter occupies only $1/2$ slice, which is $1/4^{th}$ of a CLB. THLS tool offers almost 16 times improvement in silicon efficiency than the Icarus tool.

Table 5. gives details of total equivalent gate count based on map reports generated after technology mapping in ISE. THLS tool has 1.7 times improvement in silicon efficiency over the Icarus tool. Both THLS and ISE have same silicon efficiency in this case. *ISE infer, first, a set of registers as Icarus. Later at low level synthesis only, it infers a shift register. But THLS infers it correctly at the elaboration itself.*

## VII. CONCLUSION

It is imperative that HLS must be aware of the target technology for optimal hardware generation. Towards that we suggested a new hardware compilation methodology called THLS. We experimented this methodology for few examples and generated optimal hardware. It achieves this optimization at the cost of portability.

## ACKNOWLEDGMENT

This work is funded by MHRD, Government of India.

## REFERENCES

[1] Knuth D.E, "Semantics of Context Free Languages", Mathematics Systems Theory, Vol 2, No 2, pp. 127-145, 1968.
[2] A.V.Aho, R.Sethi and J.D.Ullman,"Compilers: Principles, Techniques and Tools", *Addison-Wesley*, 1986.
[3] S.D.Brown, R.J.Francis, J.Rose and Z.G.Vranesic, "Field Programmable Gate Arrays", *Kluwer Academic Publishers*, 1992.
[4] J.P.Weng and A.C. Parker, "3D Scheduling: High-Level Synthesis with: *Proceedings of Design Automation Conference*, 1992.
[5] M.C.McFarland, A.C.Parker and R.Campasona,"Tutorial on High-Level Synthesis", *25th ACM/IEEE Design Automation Conference*, 1988.
[6] D.D.Gajski, N.D.Dutt, A.Wu and S.Lin,"High-Level Synthesis: Introduction to Chip and System Design", *Kluwer Academic Publishers*, 1992.
[7] Daniel D Gajski and Loganath Ramachandran,"Introduction to High-Level Synthesis", *IEEE Design and Test of Computers*, 1994.

[8] Y.M.Fang and D.F.Wong, “Simultaneous functional unit binding and floor planning”, *Proceedings of International Conference on Computer Aided Design*, 1994.

[9] K.Chao and D.F.Wong, Floor planning for Low power designs, *IEEE Transactions on VLSI systems*, 1995.

[10] J.M.Chang and Pedram, “Register allocation and binding for low power”, *Proceedings of Design Automation Conference*, 1995.

[11] R.Mehra, L.M.Guerra and J.M.Rabaey, “Low-power architectural synthesis and impact of exploiting locality”, *Journal on VLSI Signal Processing*, 1996.

[12] Y.L.Lin,“Recent Developments in High-Level Synthesis,” *ACM Transactions on Design Automation of Electronic Systems*, Vol. 2, No.1, pp. 2-21, 1997.

[13] A.Ragunathan and N.K.Jha, SCALP: An iterative improvement-based low power data path synthesis system, *Proceedings of International Conference on Computer Aided Design*, 1997.

[14] P.Prabhakaran and P.Banerjee, “Simultaneous scheduling, binding and floor planning in HLS”, *Proceedings of International Conference on VLSI Design*, 1998.

[15] Min Xuy, Fadi J. Kurdahi z, “Layout-Driven High Level Synthesis for FPGA Based Architectures”, *IEEE*, 1998.

[16] Oliver Bringmann, Carsten Menn, Wolfgang Rosenstie, “Target Architecture Oriented High-Level Synthesis for Multi-FPGA Based Emulation”, *DATE 2000*, 2000.

[17] L.Kruse, E.Schmidt, G.v.Colln, A. Stammermann, A.Schulz, E.Macii and W.Nebel, “Estimation of lower and upper bounds on the power consumption from scheduled data flow graphs”, *IEEE Transactions on VLSI systems*, 2001.

[18] L.Zhong and N.K.Jha, “Interconnect-aware High-Level Synthesis for Low Power”, *ICCAD*, 2002.

[19] Junhyung Um and Taewhan Kim, “Layout-Aware Synthesis of Arithmetic Circuits”, *DAC02*, ACM, 2002.

[20] A.Stammermann, D.Helms, M.Schulte and A.Schulz, W.Nebel, “Binding, allocation, floor planning in low-power High-Level Synthesis”, *Proceedings of International Conference on Computer Aided Design (ICCAD03)*, 2003.

[21] Gwenole Corre, Eric Senn, Nathalie Julin, Eric Martin, “A Memory Aware Behavioral Synthesis Tool for Real-Time VLSI Circuits”, *GLSVLSI04, ACM*, 2004.

[22] Jason Cong, Yiping Fan, Guoling Han, Yizhou Lin, Junjuan Xu1, Zhiru Zhang, Xu Cheng, “Bitwidth-Aware Scheduling and Binding in High-Level Synthesis”, *ASP_DAC 2005, IEEE*, 2005.

[23] M.Joseph, Narasimha B Bhat and K Chandra Sekaran, Survey of HDL Compiler Optimization Techniques, *International Journal on Information Processing*, Volume 1, Number 1, pp. 51 - 62, April 2007.

[24] M.Joseph, Narasimha B.Bhat and K.Chandra Sekaran, Right inference of Hardware in High-Level Synthesis, *International Conference on Information processing - ICIP 2007*, Bangalore, India, August 2007.

[25] www.edif.org

[26] www.xilinx.com

[27] www.icraus.com